Procedural Reflection in Programming Languages

Volume I

Brian Cantwell Smith

B.S., Massachusetts Institute of Technology (1974)
M.S., Massachusetts Institute of Technology (1978)

Submitted in partial fulfillment
of the requirements for the Degree of

Doctor of Philosophy

at the

Massachusetts Institute of Technology

February 1982

© Massachusetts Institute of Technology 1982

Signature of Author _____
Department of Electrical Engineering and Computer Science
January 25, 1982

Certified by _____
Peter Szolovits
Thesis Supervisor

Accepted by _____
Arthur C. Smith
Chairman, Departmental Graduate Committee

Eng.
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
MAY 20 1982
LIBRARIES

MASS. INST. OF TECHNOLOGY
MAR 09 1983
BARKER ENGINEERING LIBRARY

Abstract

We show how a computational system can be constructed to "reason", effectively and consequentially, about its own inferential processes. The analysis proceeds in two parts. First, we consider the general question of computational semantics, rejecting traditional approaches, and arguing that the *declarative* and *procedural* aspects of computational symbols (what they stand for, and what behaviour they engender) should be analysed *independently*, in order that they may be coherently related. Second, we investigate *self-referential* behaviour in computational processes, and show how to embed an effective procedural model of a computational calculus within that calculus (a model not unlike a meta-circular interpreter, but connected to the fundamental operations of the machine in such a way as to provide, at any point in a computation, fully articulated descriptions of the state of that computation, for inspection and possible modification). In terms of the theories that result from these investigations, we present a general architecture for *procedurally reflective* processes, able to shift smoothly between dealing with a given subject domain, and dealing with their own reasoning processes over that domain.

An instance of the general solution is worked out in the context of an applicative language. Specifically, we present three successive dialects of LISP: 1-LISP, a distillation of current practice, for comparison purposes; 2-LISP, a dialect constructed in terms of our rationalised semantics, in which the concept of evaluation is rejected in favour of independent notions of *simplification* and *reference*, and in which the respective categories of notation, structure, semantics, and behaviour are strictly aligned; and 3-LISP, an extension of 2-LISP endowed with reflective powers.

This research was supported (in part) by the National Institutes of Health Grant No. 1 P01 LM 03374 from the National Library of Medicine.

Extended Abstract

We show how a computational system can be constructed to "reason" effectively and consequentially about its own inference processes. Our approach is to analyse *self-referential* behaviour in computational systems, and to propose a theory of *procedural reflection* that enables any programming language to be extended in such a way as to support programs able to access and manipulate structural descriptions of their own operations and structures. In particular, one must encode an explicit theory of such a system within the structures of the system, and then connect that theory to the fundamental operations of the system in such a way as to support three primitive behaviours. First, at any point in the course of a computation fully articulated descriptions of the state of the reasoning process must be available for inspection and modification. Second, it must be possible at any point to resume an arbitrary computation in accord with such (possibly modified) theory-relative descriptions. Third, procedures that reason with descriptions of the processor state must themselves be subject to description and review, to arbitrary depth. Such *reflective* abilities allow a process to shift smoothly between dealing with a given subject domain, and dealing with its own reasoning processes over that domain.

Crucial in the development of this theory is a comparison of the respective semantics of programming languages (such as LISP and ALGOL) and declarative languages (such as logic and the λ-calculus); we argue that unifying these traditionally separate disciplines clarifies both, and suggests a simple and natural approach to the question of procedural reflection. More specifically, the semantical analysis of computational systems should comprise independent formulations of *declarative import* (what symbols stand for) and *procedural consequence* (what effects and results are engendered by processing them), although the two semantical treatments may, because of side-effect interactions, have to be formulated in conjunction. When this approach is applied to a functional language it is shown that the traditional notion of evaluation is confusing and confused, and must be rejected in favour of independent notions of *reference* and *simplification.* In addition, we defend a standard of *category alignment:* there should be a systematic correspondence between the respective categories of notation, abstract structure, declarative semantics, and procedural consequence (a mandate satisified by no extant procedural formalism). It is shown how a clarification of these prior semantical and aesthetic issues enables a *procedurally reflective* dialect to be clearly defined and readily constructed.

An instance of the general solution is worked out in the context of an applicative language, where the question reduces to one of defining an interpreted calculus able to inspect and affect its own interpretation. In particular, we consider three successive dialects of LISP: 1-LISP, a distillation of current practice for comparison purposes, 2-LISP, a dialect *categorically* and *semantically rationalised* with respect to an explicit theory of declarative semantics for s-expressions, and 3-LISP, a derivative of 2-LISP endowed with full reflective powers. 1-LISP, like all LISP dialects in current use, is at heart a *first-order* language, employing meta-syntactic facilities and dynamic variable scoping protocols to partially mimic higher-order functionality. 2-LISP, like SCHEME and the λ-calculus, is higher-order: it supports arbitrary function designators in argument position, is lexically scoped, and treats

the function position of an application in a standard extensional manner. Unlike SCHEME, however, the 2-LISP processor is based on a regimen of *normalisation*, taking each expression into a normal-form co-designator of its referent, where the notion of *normal-form* is in part defined with respect to that referent's semantic type, not (as in the case of the λ-calculus) solely in terms of the further non-applicability of a set of syntactic reduction rules. 2-LISP normal-form designators are environment-independent and side-effect free; thus the concept of a *closure* can be reconstructed as a *normal-form function designator.* In addition, since normalisation is a form of simplification, and is therefore *designation-preserving*, meta-structural expressions are not de-referenced upon normalisation, as they are when evaluated. Thus we say that the 2-LISP processor is *semantically flat*, since it stays at a semantically fixed level (although explicit referencing and de-referencing primitives are also provided, to facilitate explicit level shifts). Finally, because of its category alignment, *argument objectification* (the ability to apply functions to a sequence of arguments designated collectively by a single term) can be treated in the 2-LISP base-level language, without requiring resort to meta-structural machinery.

3-LISP is straightforwardly defined as an extension of 2-LISP, with respect to an explicitly articulated procedural theory of 3-LISP embedded in 3-LISP structures. This embedded theory, called the *reflective model*, though superficially resembling a meta-circular interpreter, is causally connected to the workings of the underlying calculus in crucial and primitive ways. Specifically, *reflective procedures* are supported that bind as arguments (designators of) the continuation and environment structure of the processor that would have been in effect at the moment the reflective procedure was called, had the machine been running all along in virtue of the explicit processing of that reflective model. Because reflection may recurse arbitrarily, 3-LISP is most simply defined as an infinite tower of 3-LISP processes, each engendering the process immediately below it. Under such an account, the use of reflective procedures amounts to running programs at arbitrary levels in this reflective hierarchy. Both a straightforward implementation and a conceptual analysis are provided to demonstrate that such a machine is nevertheless finite.

The 3-LISP reflective model unifies three programming language concepts that have formerly been viewed as independent: meta-circular interpreters, explicit names for the primitive interpretive procedures (EVAL and APPLY in standard LISP dialects), and procedures that access the state of the implementation (typically provided, as part of a programming environment, for debugging purposes). We show how all such behaviours can be defined within a pure version of 3-LISP (i.e., independent of implementation), since all aspects of the state of any 3-LISP process are available, with sufficient reflection, as objectified entities within the 3-LISP structural field.

Summary Contents

Abstract	Page 2
Extended Abstract	3
Summary Contents	5
Contents	6
Preface and Acknowledgements	10
Prologue	13
1. Introduction	26
2. 1-LISP: A Basis Dialect	103
3. Semantic Rationalisation	122
4. 2-LISP: A Rationalised Dialect	253
5. Procedural Reflection and 3-LISP	571
6. Conclusion	700
Appendix: A MACLISP Implementation of 3-LISP	707
Notes and References	752

Contents

Preliminaries **Page 2**

Abstract 2
Extended Abstract 3
Summary Contents 5
Contents 6
Preface and Acknowledgments 10

Prologue 13

Chapter 1. Introduction 26

Introduction 26
1.a. General Overview 27
1.b. The Concept of Reflection 35
1.b.i. The Reflection and Representation Hypotheses 35
1.b.ii. Reflection in Computational Formalisms 38
1.b.iii. Six General Properties of Reflection 42
1.b.iv. Reflection and Self-Reference 47
1.c. A Process Reduction Model of Computation 50
1.d. The Rationalisation of Computational Semantics 59
1.d.i. Pre-Theoretic Assumptions 59
1.d.ii. Semantics in a Computational Setting 61
1.d.iii. Recursive and Compositional Formulations 67
1.d.iv. The Role of a Declarative Semantics 69
1.e. Procedural Reflection 70
1.e.i. A First Sketch 70
1.e.ii. Meta-Circular Processors 72
1.e.iii. Procedural Reflection Models 75
1.e.iv. Two Views of Reflection 78
1.e.v. Some General Comments 80
1.f. The Use of LISP as an Explanatory Vehicle 82
1.f.i. 1-LISP as a Distillation of Current Practice 83
1.f.ii. The Design of 2-LISP 86
1.f.iii. The Procedurally Reflective 3-LISP 91
1.f.iv. Reconstruction Rather Than Design 93

1.g. Remarks 95
1.g.i. Comparison with Other Work 95
1.g.ii. The Mathematical Meta-Language 101
1.g.iii. Examples and Implementation 102

Chapter 2. 1-LISP: A Basis Dialect 103

Chapter 3. Semantic Rationalisation 122

Introduction 122
3.a. The Semantics of Traditional Systems 124
3.a.i. Logic 124
3.a.ii. The λ-Calculus 127
3.a.iii. PROLOG 129
3.a.iv. Commonalities 130
3.b. The Semantics of Computational Calculi 134
3.b.i. Standard Programming Language Semantics 134
3.b.ii. Declarative Semantics in LISP 143
3.b.iii. Summary 148
3.c. Preparations for 1-LISP and 1.7-LISP Semantics 150
3.c.i. Local and Full Procedural Consequence 150
3.c.ii. Declarative Semantics for Data Structures 153
3.c.iii. Recursive Compositionality, Extensionality, and Accessibility 155
3.c.iv. Structure vs. Notation 158
3.c.v. Context Relativity 160
3.c.vi. Terminology and Standard Models 166
3.c.vii. Declarative Semantics and Assertional Force 168
3.d. The Semantics of 1-LISP: First Attempt 170
3.d.i. Declarative Semantics (Φ) 171
3.d.ii. Local Procedural Semantics (Ψ) 190
3.d.iii. Full Procedural Semantics (Γ) 199
3.e. The Semantics of 1-LISP: Second Attempt 205
3.e.i. The Pervasive Influence of Evaluation 205
3.e.ii. The Temporal Context of Designation 207
3.e.iii. Full Computational Significance (Σ) 211
3.e.iv. An Example 218
3.e.v. The Evaluation Theorem 225
3.f. Towards a Rationalised Design 228
3.f.i. Evaluation Considered Harmful 230
3.f.ii. Normal Form Designators 237
3.f.iii. Lessons and Observations 243
3.f.iv. Declarative Import, Implementation, and Data Abstraction 246

Chapter 4. 2-LISP: A Rationalised Dialect 253

Introduction 253

4.a. The 2-LISP Structural Field 257

4.a.i. Numerals and Numbers 257
4.a.ii Booleans and Truth-values 259
4.a.iii Atoms 260
4.a.iv. Pairs and Reductions 261
4.a.v. Rails and Sequences 265
4.a.vi. Handles 286
4.a.vii. Category Summary 291
4.a.viii. Normal-form Designators 292
4.a.ix. Accessibility 297
4.a.x. Graphical Notation 298

4.b. Simple 2-LISP Primitives 301

4.b.i. Arithmetic Primitives 302
4.b.ii. Selectors on Pairs 314
4.b.iii. Typing and Identity 321
4.b.iv. Selectors on Rails and Sequences 330
4.b.v. The Creation of New Structure 335
4.b.vi. Vector Generalisations 342
4.b.vii. Structural Field Side Effects 350
4.b.viii. Input/Output 356
4.b.ix. Control 360

4.c. Methods of Composition and Abstraction 377

4.c.i. Lambda Abstraction and Procedural Intension 377
4.c.ii. Closures: Normal Form Function Designators 393
4.c.iii. Patterns and Parameter Binding 401
4.c.iv. The Semantics of LAMBDA, EXPR, and IMPR 412
4.c.v. Recursion 427
4.c.vi. Environments and the Setting of Variables 461

4.d. Meta-Structural Capabilities 481

4.d.i. NAME and REFERENT 481
4.d.ii. NORMALISE and REDUCE 493
4.d.iii. Intensional Procedures 505
4.d.iv. The "Up-Down" Theorem 512
4.d.v. Macros and Backquote 522
4.d.vi. The Normalisation ("Flat") and Type Theorems 544
4.d.vii. The 2-LISP Meta-Circular Processor 550

4.e. Conclusion 565

Chapter 5. Procedural Reflection and 3-LISP 571
Introduction 571
5.a. The Architecture of Reflection 576
5.a.i. The Limitations of 2-LISP 576
5.a.ii. Some Untenable Proposals 583
5.a.iii. Reflective Code in the Processor 595
5.a.iv. Four Grades of Reflective Involvement 600
5.b. An Introduction to 3-LISP 606
5.b.i. Reflective Procedures and Reflective Levels 608
5.b.ii. Some Elementary Examples 614
5.b.iii. LAMBDA, and Simple and Reflective Closures 621
5.b.iv. The Structure of Environments 626
5.b.v. Simple Debugging 633
5.b.vi. REFERENT 638
5.b.vii. The Conditional 641
5.b.viii. Review and Comparision with 2-LISP 645
5.c. The Reflective Processor 648
5.c.i. The Integration of Reflective Procedures 649
5.c.ii. The Treatment of Primitives 652
5.c.iii. Levels of READ-NORMALISE-PRINT 656
5.c.iv. Control Flow in the Reflective Processor 661
5.c.v. The Implementation of a Reflective Dialect 671
5.d. Reflection in Practice 679
5.d.i. Continuations with a Variable Number of Arguments 679
5.d.ii. Macros 688
5.d.iii. Pointers to Further Examples 695
5.e. The Mathematical Characterisation of Reflection 699

Chapter 6. Conclusion 700

Appendix. A MACLISP Implementation of 3-LISP 707

Notes 752
References 756

Preface and Acknowledgements

The possibility of constructing a reflective calculus first struck me in June 1976, at the Xerox Palo Alto Research Center, where I was spending a summer working with the KRL representation language of Bobrow and Winograd.[1] As an exercise to learn the language, I had embarked on the project of representing KRL in KRL; it seemed to me that this "double-barrelled" approach, in which I would have both to *use* and to *mention* the language, would be a particularly efficient way to unravel its intricacies. Though that exercise was ultimately abandoned, I stayed with it long enough to become intrigued by the thought that one might build a system that was self-descriptive in an important way (certainly in a way in which my KRL project was *not*). More specifically, I could dimly envisage a computational system in which what happened took effect in virtue of declarative descriptions of what was to happen, and in which the internal structural conditions were represented in declarative descriptions of those internal structural conditions. In such a system a program could with equal ease access all the basic operations and structures either directly or in terms of completely (and automatically) articulated descriptions of them. The idea seemed to me rather simple (as it still does); furthermore, for a variety of reasons I thought that such a reflective calculus could *itself* be rather simple — in some important ways simpler than a non-reflective formalism (this too I still believe). *Designing* such a formalism, however, no longer seems as straightforward as I thought at the time; this dissertation should be viewed as the first report emerging from the research project that ensued.

Most of the five years since 1976 have been devoted to initial versions of my specification of such a language, called MANTIQ, based on these original hunches. As mentioned in the first paragraph of chapter 1, there are various non-trivial goals that must be met by the designer of any such formalism, including at least a tentative solution to the knowledge representation problem. Furthermore, in the course of its development, MANTIQ has come to rest on some additional hypotheses above and beyond those mentioned above (including, for example, a sense that it will be possible within a computational setting to construct a formalism in which syntactic identity and intensional identity can be identified, given some appropriate, but independently specified, theory of intensionality). Probably

the major portion of my attention to date has focused on these intensional aspects of the MANTIQ architecture.

It was clear from the outset that no dialect of LISP (or of any other purely procedural calculus) could serve as a full reflective formalism; purely declarative languages like logic or the λ-calculus were dismissed for similar reasons. In February of 1981, however, I decided that it would be worth focusing on LISP, by way of an example, in order to work out the details of a specific subset of the issues with which MANTIQ would have to contend. In particular, I recognised that many of the questions of reflection could be profitably studied in a (limited) procedural dialect, in ways that would ultimately illuminate the larger programme. Furthermore, to the extent that LISP could serve as a theoretical vehicle, it seemed a good project; it would be much easier to develop, and even more so to communicate, solutions in a formalism at least partially understood.

The time from the original decision to look at procedural reflection (and its concomitant emphasis on semantics — I realised from investigations of MANTIQ that semantics would come to the fore in all aspects of the overall enterprise), to a working implementation of 3-LISP, was only a few weeks. Articulating why 3-LISP was the way it was, however — formulating in plain English the concepts and categories on which the design was founded — required quite intensive work for the remainder of the year. A first draft of the dissertation was completed at the end of December 1981; the implementation remained essentially unchanged during the course of this writing (the only substantive alteration was the idea of treating recursion in terms of explicit Y operators). Thus (and I suspect there is nothing unusual in this experience) formulating an idea required approximately ten times more work than embodying it in a machine; perhaps more surprisingly, all of that effort in formulation occurred *after* the implementation was complete. We sometimes hear that writing computer programs is intellectually hygenic because it requires that we make our ideas completely explicit. I have come to disagree rather fundamentally with this view. Certainly writing a program does not force one to one make one's ideas *articulate*, although it is a useful first step. More seriously, however, it is often the case that the organising principles and fundamental insights contributing to the coherence of a program are not explicitly encoded within the structures comprising that program. The theory of declarative semantics embodied in 3-LISP, for example, was initially tacit — a fact perhaps to be expected, since only procedural consequence is

explicitly encoded in an implementation. Curiously, this is one of the reasons that building a fully reflective formalism (as opposed to the limited procedurally reflective languages considered here) is difficult: in order to build a general reflective calculus, one must embed within it a fully articulated theory of one's understanding of it. This will take some time.

An itinerant graduate student career has made me indelibly indebted to more people than can possibly be named here. It is often pointed out that any ideas or contributions that a person makes arise not from the individual, but from the embedding context and community within which he or she works; this is doubly true when the project — as is the case here — is one of rational reconstruction. It is the explicit intent of this dissertation to articulate the tacit conception of programming that we all share; thus I want first to acknowledge the support and contributions of all those attempting to develop and to deploy the overarching computational metaphor.

Particular thanks are due to my committee members: Peter Szolovits, Terry Winograd, and Jon Allen, not only for the time and judgment they gave to this particular dissertation, but also for their sustaining support over many years, through periods when it was clear to none of us how (or perhaps even whether) I would be able to delineate and concentrate on any finite part of the encompassing enterprise. I am grateful as well to Terry Winograd and Danny Bobrow for inviting me to participate in the KRL project where this research began, and to them and to my fellow students in that research group (David Levy, Paul Martin, Mitch Model, and Henry Thompson) for their original and continued support.

Finally, in the years between that seminal summer and the present, any number of people have contributed to my understanding and commitment, in ways that they alone know best. Let me appreciatively just mention my family, and Bob Berwick, Ron Brachman, John Brown, Chip Bruce, Dedre Gentner, Barbara Grosz, Austin Henderson, David Israel, Marcia Lind, Mitch Marcus, Marilyn Matz, Ray Perrault, Susan Porter, Bruce Roberts, Arnold Smith, Al Stevens, Hector LeVesque, Sylvia Weir, and again Terry Winograd.

Prologue

It is a striking fact about human cognition that we can think not only about the world around us, but also about our ideas, our actions, our feelings, our past experience. This ability to *reflect* lies behind much of the subtlety and flexibility with which we deal with the world; it is an essential part of mastering new skills, of reacting to unexpected circumstances, of short-range and long-range planning, of recovering from mistakes, of extrapolating from past experience, and so on and so forth. Reflective thinking characterises mundane practical matters and delicate theoretical distinctions. We have all paused to review past circumstances, such as conversations with guests or strangers, to consider the appropriateness of our behaviour. We can remember times when we stopped and consciously decided to consider a set of options, say when confronted with a fire or other emergency. We understand when someone tells us to believe everything a friend tells us, unless we know otherwise. In the course of philosophical discussion we can agree to distinguish views we believe to be true from those we have no reason to believe are false. In all these cases the subject matter of our contemplation at the moment of reflection includes our remembered experience, our private thoughts, and our reasoning patterns.

The power and universality of reflective thinking has caught the attention of the cognitive science community — indeed, once alerted to this aspect of human behaviour, theorists find evidence of it almost everywhere. Though no one can yet say just what it comes to, crucial ingredients would seem to be the ability to recall memories of a world experienced in the past and of one's own participation in that world, the ability to think about a phenomenal world, hypothetical or actual, that is not currently being experienced (an ability presumably mediated by our knowledge and belief), and a certain kind of true self-reference: the ability to consider both one's actions and the workings of one's own mind. This last aspect — the self-referential aspect of reflective thought — has sparked particular interest for cognitive theorists, both in psychology (under the label *meta-cognition*), and in artificial intelligence (in the design of computational systems possessing inchoate reflective powers, particularly as evidenced in a collection of ideas loosely allied in their use of the term "meta": meta-level rules, meta-descriptions, and so forth).

In artificial intelligence, the focus on computational forms of self-referential reflective reasoning has become particularly central. Although the task of endowing computational systems with subtlety and flexibility has proved difficult, we have had some success in developing systems with a moderate grasp of certain domains: electronics, bacteremia, simple mechanical systems, etc. One of the most recalcitrant problems, however, has been that of developing flexibility and modularity (in some cases even simple effectiveness) in the reasoning processes that use this world knowledge. Though it has been possible to construct programs that perform a specific kind of reasoning task (say, checking an circuit or parsing a subset of natural language syntax), there has been less success in simulating "common sense", or in developing programs able to figure out what to do, and how to do it, in either general or novel situations. If the course of reasoning — if the problem solving strategies and the hypothesis formation behaviour — could *itself* be treated as a valid subject domain in its own right, then (at least so the idea goes) it might be possible to construct systems that manifested the same modularity about their own thought processes that they manifest about their primary subject domains. A simple example might be an electronics "expert" able to choose an appropriate method of tackling a particular circuit, depending on a variety of questions about the relationship between its own capacities and the problem at hand: whether the task was primarily one of design or analysis or repair, what strategies and skills it knew it had in such areas, how confident it was in the relevance of specific approaches based on, say, the complexity of the circuit, or on how similar it looked compared with circuits its already knew. Expert human problem-solvers clearly demonstrate such reflective abilities, and it appears more and more certain that powerful computational problem solvers will have to possess them as well.

No one would expect potent skills to arise automatically in a reflective system; the mere *ability* to reason about the reasoning process will not magically yield systems able to reflect in powerful and flexible ways. On the other hand, the demonstration of such an ability is clearly a pre-requisite to its effective utilisation. Furthermore, many reasons are advanced in support of reflection, as well as the primary one (the hope of building a system able to decide how to structure the pattern of its own reasoning). It has been argued, for example, that it would be easier to construct powerful systems in the first place (it would seem you could almost *tell them* how to think), to interact with them when they fail, to trust them if they could report on how they arrive at their decisions, to give them "advice"

about how to improve or discriminate, as well as to provide them with their own strategies for reacting to their history and experience.

There is even, as part of the general excitement, a tentative suggestion on how such a self-referential reflective process might be constructed. This suggestion — nowhere argued but clearly in evidence in several recent proposals — is a particular instance of a general hypothesis, adopted by most A.I. researchers, that we will call the ***knowledge representation hypothesis***. It is widely held in computational circles that any process capable of reasoning intelligently about the world must consist in part of a field of structures, of a roughly linguistic sort, which in some fashion *represent* whatever knowledge and beliefs the process may be said to possess. For example, according to this view, since I know that the sun sets each evening, my "mind" must contain (among other things) a language-like or symbolic structure that represents this fact, inscribed in some kind of internal code. There are various assumptions that go along with this view: there is for one thing presumed to be an internal process that "runs over" or "computes with" these representational structures, in such a way that the intelligent behaviour of the whole results from the interaction of parts. In addition, this ingredient process is required to react only to the "form" or "shape" of these mental representations, without regard to what they mean or represent — this is the substance of the claim that computation involves *formal* symbol manipulation. Thus my thought that, for example, the sun will soon set, would be taken to emerge from an interaction in my mind between an ingredient process and the shape or "spelling" of various internal structures representing my knowledge that the sun does regularly set each evening, that it is currently tea time, and so forth.

The knowledge representation hypothesis may be summarised as follows:

> *Any mechanically embodied intelligent process will be comprised of structural ingredients that a) we as external observers naturally take to represent a propositional account of the knowledge that the overall process exhibits, and b) independent of such external semantical attribution, play a formal but causal and essential role in engendering the behaviour that manifests that knowledge.*

Thus for example if we felt disposed to say that some process knew that dinosaurs were warm-blooded, then we would find (according, presumably, to the best explanation of how that process worked) that a certain computational ingredient in that process was understood as *representing* the (propositional) fact that dinosaurs were warm-blooded, and furthermore

that this very ingredient played a role, independent of our understanding of it as representational, in leading the process to behave in whatever way inspired us to say that it knew that fact. Presumably we would convinced by the manner in which the process answered certain questions about their likely habitat, by assumptions it made about other aspects of their existence, by postures it adopted on suggestions as to why they may have become extinct, etc.

A careful analysis will show that, to the extent that we can make sense of it, this view that *knowing is representational* is far less evident — and perhaps, therefore, far more interesting — than is commonly believed. To do it justice requires considerable care: accounts in cognitive psychology and the philosophy of mind tend to founder on simplistic models of computation, and artificial intelligence treatments often lack the theoretical rigour necessary to bring the essence of the idea into plain view. Nonetheless, conclusion or hypothesis, it permeates current theories of mind, and has in particular led researchers in artificial intelligence to propose a spate of computational languages and calculi designed to underwrite such representation. The common goal is of course not so much to speculate on what is actually represented in any particular situation as to uncover the general and categorical form of such representation. Thus no one would suggest how anyone actually represents facts about tea and sunsets: rather, they might posit the general form in which such beliefs would be "written" (along with other beliefs, such as that Lasa is in Tibet, and that π is an irrational number). Constraining all plausible suggestions, however, is the requirement that they must be able to demonstrate how a particular thought could emerge from such representations — this is a crucial meta-theoretic characteristic of artificial intelligence research. It is traditionally considered insufficient merely to propose true theories that do not enable some causally effective mechanical embodiment. The standard against which such theories must ultimately judged, in other words, is whether they will serve to underwrite the construction of demonstrable, behaving artefacts. Under this general rubric knowledge representation efforts differ markedly in scope, in approach, and in detail; they differ on such crucial questions as whether or not the mental structure are modality specific (one for visual memory, another for verbal, for example). In spite of such differences, however, they manifest the shared hope that an attainable first step towards a full theory of mind will be the discovery of something like the structure of the "mechanical mentalese" in which our beliefs are inscribed.

It is natural to ask whether the knowledge representation hypothesis deserves our endorsement, but this is not the place to pursue that difficult question. Before it can fairly be asked, we would have to distinguish a strong version claiming that knowing is *necessarily* representational from a weaker version claiming merely that it is *possible* to build a representational knower. We would run straight into all the much-discussed but virtually intractable questions about what would be required to convince us that an artificially constructed process exhibited intelligent behaviour. We would certainly need a definition of the word "represent", about which we will subsequently have a good deal to say. Given the current (minimal) state of our understanding, I myself see no reason to subscribe to the strong view, and remain skeptical of the weak version as well. But one of the most difficult questions is merely to ascertain what the hypothesis is actually saying — thus my interest in representation is more a concern to make it clear than to defend or deny it. The entire present investigation, therefore, will be pursued under this hypothesis, not because we grant it our allegiance, but merely because it deserves our attention.

Given the represention hypothesis, the suggestion as to how to build self-reflective systems — a suggestion we will call the ***reflection hypothesis*** — can be summarised as follows:

> *In as much as a computational process can be constructed to reason about an external world in virtue of comprising an ingredient process (interpreter) formally manipulating representations of that world, so too a computational process could be made to reason about itself in virtue of comprising an ingredient process (interpreter) formally manipulating representations of its own operations and structures.*

Thus the task of building a computationally reflective system is thought to reduce to, or at any rate to include, the task of providing a system with formal representations of its own constitution and behaviour. Hence a system able to imagine a world where unicorns have wings would have to construct formal representations of that fact; a system considering the adoption of a hypothesis-and-test style of investigation would have to construct formal structures representing such a inference regime.

Whatever its merit, there is ample evidence that researchers are taken with this view. Systems such as Weyrauch's FOL, Doyle's TMS, McCarthy's ADVICE-TAKER, Hayes' GOLUM, and Davis' TERESIUS are particularly explicit exemplars of just such an approach.[2] In

Weyhrauch's system, for example, sentences in first-order logic are constructed that axiomatize the behaviour of the LISP procedures use in the course of the computation (FOL is a prime example of the dual-calculus approach mentioned earlier). In Doyle's systems, explicit representations of the dependencies between beliefs, and of the "reasons" the system accepts a conclusion, play a causal role in the inferential process. Similar remarks hold for the other projects mentioned, as well as for a variety of other current research. In addition, it turns out on scrutiny that a great deal of current computational practice can be seen as dealing, in one way or another, with reflective abilities, particularly as exemplified by computational structures representing other computational structures. We constantly encounter examples: the wide-spread use of macros in LISP, the use of meta-level structures in representation languages, the use of explicit non-monotonic inference rules, the popularity of meta-level rules in planning systems.[3] Such a list can be extended indefinitely; in a recent symposium Brachman reported that the love affair with "*meta-level reasoning*" was the most important theme of knowledge representation research in the last decade.[4]

The Relationship Between Reflection and Representation

The manner in which this discussion has been presented so far would seem to imply that the interest in *reflection* and the adoption of a *representational* stance are theoretically independent positions. I have argued in this way for a reason: to make clear that the two subjects are not the same. There is no *a priori* reason to believe that even a fully representational system should in any way be reflective or able to make anything approximating a reference to itself; similarly, there is no *proof* that a powerfully self-referential system need be constructed of representations. However — and this is the crux of the matter — the reason to raise both issues together is that they are surely, in some sense, related. If nothing else, the word "representation" comes from "re" plus "present", and the ability to *re-present* a world to itself is undeniably a crucial, if not *the* crucial, ingredient in reflective thought. If I reflect on my childhood, I re-present to myself my school and the rooms of my house; if I reflect on what I will do tomorrow, I bring into the view of my mind's eye the self I imagine that tomorrow I will be. If we take "representation" to describe an *activity*, rather than a *structure*, reflection surely involves representation (although — and this should be kept clearly in mind — the "representation"

of the knowledge representation hypothesis refers to ingredient structures, not to an activity).

It is helpful to look at the historical association between these ideas, as well to search for commonalities in content. In the early days of artificial intelligence, a search for the general patterns of intelligent reasoning led to the development of such general systems as Newell and Simon's GPS, predicate logic theorem provers, and so forth.[5] The descriptions of the subject domains were minimal but were nonetheless primarily declarative, particularly in the case of the systems based on logic. However it proved difficult to make such general systems effective in particular cases: so much of the "expertise" involved in problem solving seems domain and task specific. In reaction against such generality, therefore, a *procedural* approach emerged in which the primary focus was on the manipulation and reasoning about specific problems in simple worlds.[6] Though the procedural approach in many ways solved the problem of undirected inferential meandering, it too had problems: it proved difficult to endow systems with much generality or modularity when they were simply constituted of procedures designed to manifest certain particular skills. In reaction to such brittle and parochial behaviour, researchers turned instead to the development of processes designed to work over general representations of the objects and categories of the world in which the process was designed to be embedded. Thus the *representation hypothesis* emerged in the attempt to endow systems with generality, modularity, flexibility, and so forth with respect to the embedding world, but to retain a procedural effectiveness in the control component.[7] In other words, in terms of our main discussion, representation as a method emerged as a solution to the problem of providing general and flexible ways of reflecting (not self-referentially) about the world.

Systems based on the representational approach — and it is fair to say that most of the current "expert systems" are in this tradition — have been relatively successful in certain respects, but a major lingering problem has been a narrowness and inflexibility regarding the style of reasoning these systems employ in using these representational structures. This inflexibility in *reasoning* is strikingly parallel to the inflexibility in *knowledge* that led to the first round of representational systems; researchers have therefore suggested that we need reflective systems able to deal with their own constitutions as well as with the worlds they inhabit. In other words, since the *style* of the problem is so parallel to that just sketched, it has seemed that another application of the same medicine might be

appropriate. If we could inscribe general knowledge about how to reason in a variety of circumstances in the "mentalese" of these systems, it might be possible to design a relatively simpler inferential regime over this "meta-knowledge about reasoning", thereby engendering a flexibility and modularity regarding reasoning, just as the first representational work engendered a flexibility and modularity about the process's embedding world.

There are problems, however, in too quick an association between the two ideas, not the least of which is the question of to *whom* these various forms of re-presentation are being directed. In the normal case — that is to say, in the typical computational process built under the aegis of the knowledge representation hypothesis — a process is constituted from symbols that we as external theorists take to be representational structures; they are visible *only to the ingredient interpretive process of the whole*, and they are visible to that constituent process *only formally* (this is the basic claim of computation). Thus the interpreter can see them, though it is blind to the fact of their being representations. (In fact it is almost a great joke that the blindly formal ingredient process should be called an *interpreter*: when the LISP interpreter evalutes the expression `(+ 2 3)` and returns the result `5`, the last thing it knows is that the numeral `2` denotes the number two.)

Whatever is the case with the ingredient process, there is no reason to suppose that the representational structures are visible to the whole constituted process *at all*, formally or informally. That process is made out of them; there is no more *a priori* reason to suppose that they are accessible to its inspection than to suppose that a camera could take a picture of its own shutter — no more reason to suppose it is even a coherent possibility than to say that France is near Marseilles. Current practice should overwhelmingly convince us of this point: what is as tacit — what is as thoroughly lacking in self-knowledge — as the typical modern computer system?

The point of the argument here is not to prove that one *cannot* make such structures accessible — that one *cannot* make a representational reflective system — but to make clear that two ideas are involved. Furthermore, they are different in kind: one (representation) is a possibly powerful *method* for the construction of systems; the other (reflection) is a kind of behaviour we are asking our systems to exhibit. It remains a question whether the representational method will prove useful in the pursuit of the goal of reflective behaviour.

That, in a nutshell, is our overall project.

The Theoretical Backdrop

It takes only a moment's consideration of such questions as the relationship between representation and reflection to recognise that the current state of our understanding of such subjects is terribly inadequate. In spite of the general excitement about reflection, self-reference, and computational representation, no one has presented an underlying theory of any of these issues. The reason is simple: we are so lacking in adequate theories of the surrounding territory that, without considerable preliminary work, cogent definitions cannot even be attempted. Consider for example the case regarding self-referential reflection, where just a few examples will make this clear. First, from the fact that a reflective system A is implemented in system B, it does not follow that system B is thereby rendered reflective (for example, in this dissertation I will present a partially-reflective dialect of LISP that I have implemented on a PDP-10, but the PDP-10 is not itself reflective). Hence even a *definition* of reflection will have to be backed by theoretical apparatus capable of distinguishing between one abstract machine and another in which the first is implemented — something we are not yet able to do. Second, the notion seems to require of a computational process, and (if we subscribe to the representational hypothesis) of its interpreter, that in reflecting it "back off" one level of reference, and we lack theories both of interpreters in general, and of computational reference in particular. Theories of computational interpretation will be required to clarify the confusion mentioned above regarding the relationship between reflection and representation: for a system to reflect it must re-present *for itself* its mental states; it is not sufficient for it to comprise a set of formal representations inspected *by its interpreter.* This is a distinction we encounter again and again; a failure to make it is the most common error in discussions of the plausibility of artificial intelligence from those outside the computational community, derailing the arguments of such thinkers as Searle and Fodor.[8] Theories of reference will be required in order to make sense of the question of what a computational process is "thinking" about at all, whether reflective or not (for example, it may be easy to claim that when a program is manipulating data structures representing women's vote that the process as a whole is "thinking about suffrage", but what is the process thinking about when the interpreter is expanding a macro definition?). Finally, if the search for reflection is taken up too

enthusiastically, one is in danger of interpreting everything as evidence of reflective thinking, since what may not be reflective *explicitly* can usually be treated as *implicitly* reflective (especially given a little imagination on the part of the theorist). However we lack general guidelines on how to distinguish explicit from implicit aspects of computational structures.

Nor is our grasp of the representational question any clearer; a serious difficulty, especially since the representational endeavour has received much more attention than has reflection. Evidence of this lack can be seen in the fact that, in spite of an approximate consensus regarding the general form of the task, and substantial effort on its behalf, no representation scheme yet proposed has won substantial acceptance in the field. Again, this is due at least in part to the simple absence of adequate theoretical foundations in terms of which to formulate either enterprise or solution. We do not have theories of either representation or computation in terms of which to define the terms of art currently employed in their pursuit (*representation, implementation, interpretation, control structure, data structure, inheritance,* and so forth), and are consequently without any well-specified account of what it would be to succeed, let alone of what to investigate, or of how to proceed. Numerous related theories have been developed (model theories for logic, theories of semantics for programming languages, and so forth), but they don't address the issues of knowledge representation directly, and it is surprisingly difficult to weave their various insights into a single coherent whole.

The representational consensus alluded to above, in other words, is widespread but vague; disagreements emerge on every conceivable technical point, as was demonstrated in a recent survey of the field.[9] To begin with, the central notion of "representation" remains notoriously unspecified: in spite of the intuitions mentioned above, there is remarkably little agreement on whether a representation must "re-present" in any constrained way (like an image or copy), or whether the word is synonymous with such general terms as "sign" or "symbol". A further confusion is shown by an inconsistency in usage as to what representation is a relationship between. The sub-discipline is known as the *representation of knowledge,* but in the survey just mentioned by far the majority of the respondents (to the surprise of this author) claimed to use the word, albeit in a wide variety of ways, as between formal symbols *and the world about which the process is designed to reason.* Thus a KLONE structure might be said to *represent Don Quixote tilting at a windmill*; it would not

taken as representing *the fact or proposition of this activity.* In other words the majority opinion is not that we are *representing knowledge* at all, but rather, as we put it above, that *knowing is representational.*[10]

In addition, we have only a dim understanding of the relationship that holds between the purported representational structures and the ingredient process that interprets them. This relates to the crucial distinction between that interpreting process and the whole process of which it is an ingredient (whereas it is *I* who thinks of sunsets, it is at best a *constituent of my mind* that inspects a mental representation). Furthermore, there are terminological confusions: the word "semantics" is applied to a variety of concerns, ranging from how natural language is translated into the representational structures, to what those structures represent, to how they impinge on the rational policies of the "mind" of which they are a part, to what functions are computed by the interpreting process, etc. The term "interpretation" (to take another example) has two relatively well-specified but quite independent meanings, one of computational origin, the other more philosophical; how the two relate remains so far unexplicated, although, as was just mentioned, they are strikingly distinct.

Unfortunately, such general terminological problems are just the tip of an iceberg. When we consider our specific representational proposals, we are faced with a plethora of apparently incomparable technical words and phrases. *Node, frame, unit, concept, schema, script, pattern, class,* and *plan,* for example, are all popular terms with similar connotations and ill-defined meaning.[11] The theoretical situation (this may not be so harmful in terms of more practical goals) is further hindered by the tendency for representational research to be reported in a rather demonstrative fashion: researchers typically exhibit particular formal systems that (often quite impressively) embody their insights, but that are defined using formal terms peculiar to the system at hand. We are left on our own to induce the relevant generalities and to locate them in our evolving conception of the representation enterprise as a whole. Furthermore, such practice makes comparison and discussion of technical details always problematic and often impossible, defeating attempts to build on previous work.

This lack of grounding and focus has not passed unnoticed: in various quarters one hears the suggestion that, unless severely constrained, the entire representation enterprise

may be ill-conceived — that we should turn instead to considerations of particular epistemological issues (such as how we reason about, say, liquids or actions), and should use as our technical base the traditional formal systems (logic, LISP, and so forth) that representation schemes were originally designed to replace.[12] In defense of this view two kinds of argument are often advanced. The first is that questions about the *central* cognitive faculty are at the very least premature, and more seriously may for principled reasons never succomb to the kind of rigourous scientific analysis that characterizes recent studies of the *peripheral* aspects of mind: vision, audition, grammar, manipulation, and so forth.[13] The other argument is that logic as developed by the logicians is in itself sufficient; that all we need is a set of ideas about what axioms and inference protocols are best to adopt.[14] But such doubts cannot be said to have deterred the whole of the community: the survey just mentioned lists more than thirty new representation systems under active development.

The strength of this persistence is worth noting, especially in connection with the theoretical difficulties just sketched. There can be no doubt that there are scores of difficult problems: we have just barely touched on some of the most striking. But it would be a mistake to conclude in discouragement that the *enterprise* is doomed, or to retreat to the meta-theoretic stability of adjacent fields (like proof theory, model theory, programming language semantics, and so forth). The moral is at once more difficult and yet more hopeful. What is demanded is that we stay true to these undeniably powerful ideas, and attempt to develop adequate theoretical structures on this home ground. It is true that any satisfactory theory of computational reflection must ultimately rest, more or less explicitly, on theories of computation, of intensionality, of objectification, of semantics and reference, of implicitness, of formality, of computation interpretation, of representation, and so forth. On the other hand as a community we have a great deal of practice that often embodies intuitions that we are unable to formulate coherently. The wealth of programs and systems we have built often betray — sometimes in surprising ways — patterns and insights that eluded our conscious thoughts in the course of their development. What is mandated is a *rational reconstruction* of those intuitions and of that practice.

In the case of designing reflective systems, such a reconstruction is curiously urgent. In fact this long introductory story ends with an odd twist — one that "ups the ante" in the search for a carefully formulated theory, and suggests that practical progress will be

impeded until we take up the theoretical task. In general, it is of course possible (some would even advocate this approach) to build an instance of a class of artefact before formulating a theory of it. The era of sail boats, it has often been pointed out, was already drawing to a close just as the theory of airfoils and lift was being formulated — the theory that, at least at the present time, best explains how those sailboats worked. However there are a number of reasons why such an approach may be ruled out in the present case. For one thing, in constructing a reflective calculus one must support arbitrary levels of meta-knowledge and self-modelling, and it is self-evident that confusion and complexity will multiply unckecked when one adds such faciliti . . an only partially understood formalism. It is simply likely to be unmanageably complicated to attempt to build a self-referential system unaided by the clarifying structure of a prior theory. The complexities surrounding the use of APPLY in LISP (and the caution with which it has consequently come to be treated) bear witness to this fact. However there is a more serious problem. If one subscribes to the knowledge representation hypothesis, it becomes an integral part of developing self-descriptive systems to provide, encoded within the representational medium, an account of (roughly) the syntax, semantics, and reasoning behaviour of that formalism. In other words, if we are to build a process that "knows" about itself, and *if we subscribe to the view that knowing is representational*, then we are committed to providing that sytem with a *representation* of the self-knowledge that we aim to endow it with. That is, we must have an adequate theories of computational representation and reflection *explicitly formulated*, since *an encoding of that theory is mandated to play a causal role as an actual ingredient in the reflective device.*

Knowledge of any sort — and self-knowledge is no exception — is always theory relative. The representation hypothesis implies that our theories of reasoning and reflection must be explicit. We have argued that this is a substantial, if widely accepted, hypothesis. One reason to find it plausible comes from viewing the entire enterprise as an attempt to communicate our thought patterns and cognitive styles — including our reflective abilities — to these emergent machines. It may at some point be possible for understanding to be tacitly communicated between humans and system they have constructed. In the meantime, however, while we humans might make do with a rich but unarticulated understanding of computation, representation, and reflection, we must not forget that computers do not share with us our tacit understanding of what they are.

Chapter 1. Introduction

The successful development of a general reflective calculus based on the knowledge representation hypothesis will depend on the prior solution of three problems:

1. The provision of a computationally tractable and epistemologically adequate descriptive language,
2. The formulation of a unified theory of computation and representation, and
3. The demonstration of how a computational system can reason effectively and consequentially about its own inference processes.

The first of these issues is the collective goal of present knowledge representation research; though much studied, it has met with only partial success. The problems involved are enormous, covering such diverse issues as adequate theories of intensionality, methods of indexing and grouping representational structures, and support for variations in assertional force. In spite of its centrality, however, it will not be pursued here, in part because it is so ill-constrained. The second, though it is occasionally acknowledged to be important, is a much less well publicised issue, having received (so far as this author knows) almost no direct attention. As a consequence, every representation system proposed to date exemplifies what we may call a ***dual-calculus*** approach: a procedural calculus (usually LISP) is conjoined with a declarative formalism (an encoding of predicate logic, frames, etc.). Even such purportedly unified systems as PROLOG[1] can be shown to manifest this structure. We will in passing suggest that this dual-calculus style is unnecessary and indicative of serious shortcomings in our conception of the representational endeavour. However this issue too will be largely ignored. The focus instead will be on the third problem: the question of making the inferential or interpretive aspects of a computational process themselves accessible as a valid domain of reasoning. We will show how to construct a computational system whose active interpretation is controlled by structures themselves available for inspection, modification, and manipulation, in ways that allow a process to shift smoothly between dealing with a given subject domain, and dealing with its own reasoning processes over that domain. In computational terms, the question is one of how to construct a program able to reason about and affect its own interpretation — of how to define a calculus with a reflectively accessible control structure.

1.a. General Overview

The term "reflection" does not name a previously well-defined question to which we propose a particular solution (although the *reflection principles* of logic are not unrelated); before what can present a theory of what reflection comes to, therefore, we will have to give an account of what reflection is. In the next section, by way of introduction, we will identify six distinguishing characteristics of all reflective behaviour. Then, since we will be primarily concerned with *computational* reflection, we will sketch the model of computation on which our analysis will be based, and will set our general approach to reflection into a computational context. In addition, once we have developed a working vocabulary of computational concepts, we will be able to define what we mean by *procedural* reflection — an even smaller and more circumscribed notion than computational reflection in general. All of these preliminaries are necessary in order to give us an attainable set of goals.

Thus prepared, we will set forth on the analysis itself. As a technical device, we will in the course of the dissertation develop three successive dialects of LISP, to serve as illustrations, and to provide a technical ground in which to work out our theories in detail. We should say at the outset, however, that this focus on LISP should not mislead the reader into thinking that the basic reflective architecture we will adopt — or the principles endorsed in its design — are in any important sense LISP specific. LISP was chosen because it is simple, powerful, and uniquely suited for reflection in two ways: it already embodies protocols whereby programs are represented in first-class accessible structures, and it is a convenient formalism in which to express its own meta-theory, given that we will use a variant of the λ-calculus as our mathematical meta-language (this convenience holds especially in a statically scoped dialect of the sort we will ultimately adopt). Nevertheless, as we will discuss in the concluding chapter, it would be possible to construct a reflective dialect of FORTRAN, SMALLTALK, or any other procedural calculus, by pursuing essentially the same approach as we have followed here for LISP.

The first LISP dialect (called 1-LISP) will be an example intended to summarise current practice, primarily for comparison and pedagogical purposes. The second (2-LISP) differs rather substantially from 1-LISP, in that it is modified with reference to a theory of declarative denotational semantics (i.e., a theory of the denotational significance of s-expressions) formulated *independent of the behaviour of the interpreter.* The interpreter is

then subsequently defined with respect to this theory of attributed semantics, so that the result of processing of an expression — i.e., the the value of the function computed by the basic interpretation process — is a *normal-form codesignator* of the input expression. We will call 2-LISP a *semantically rationalised* dialect, and will argue that it makes explicit much of the understanding of LISP that tacitly organises most programmers' understanding of LISP but that has never been made an arrticulated part of LISP theories. Finally, a procedurally reflective LISP called 3-LISP will be developed, semantically and structurally based on 2-LISP, but modified so that reflective procedures are supported, as a vehicle with which to engender the sorts of procedural reflection we will by then have set as our goal. 3-LISP differs from 2-LISP in a variety of ways, of which the most important is the provision, at any point in the course of the computation, for a program to *reflect* and thereby obtain fully articulated "descriptions", formulated with respect to a primitively endorsed and encoded theory, of the state of the interpretation process that was in effect at the moment of reflection. In our particular case, this will mean that a 3-LISP program will be able to access, inspect, and modify standard 3-LISP normal-form designators of both the environment and continuation structures that were in effect a moment before.

More specifically, 1-LISP, like LISP 1.5 and all LISP dialects in current use, is at heart a *first-order* language, employing meta-syntactic facilities and dynamic variable scoping protocols to partially mimic higher-order functionality. Because of its meta-syntactic powers (paradigmatically exemplified by the primitive QUOTE), 1-LISP contains a variety of inchoate reflective features, all of which we will examine in some detail: support for meta-circular interpreters, explicit names for the primitive processor functions (EVAL and APPLY), the ability to *mention* program fragments, protocols for expanding macros, and so on and so forth. Though we will ultimately criticise much of 1-LISP's structure (and its underlying theory), we will document its properties in part to serve as a contrast for the subsequent dialects, and in part because, being familiar, 1-LISP can serve as a base in which to ground our analysis.

After introducing 1-LISP, but before attempting to construct a reflective dialect, we will subject 1-LISP to a rather thorough semantical scrutiny. This project, and the reconstruction that results, will occupy well over half of the dissertation. The reason is that our analysis will require a reconstruction not only of LISP but of computational semantics in general. We will argue that it is crucial, in order to develop a comprehensible reflective

calculus, to have a semantical analysis of that calculus that makes explicit the tacit attribution of significance that we will claim characterises every computational system. This attribution of semantical import to computational expressions is *prior* to an account of what *happens* to those expressions: thus we will argue for an analysis of computational formalisms in which *declarative import* and *procedural consequence* are independently formulated. We claim, in other words, that programming languages are better understood in terms of *two* semantical treatments (one declarative, one procedural), rather than in terms of a single one, as exemplified by current approaches (although interactions between them may require that these two semantical accounts be formulated in conjunction).

This semantical reconstruction is at heart a comparison and combination of the standard semantics of programming languages on the one hand, and the semantics of natural human languages and of descriptive and declarative languages such as predicate logic, the λ-calculus, and mathematics, on the other. Neither will survive intact: the approach we will ultimately adopt is not strictly compositional in the standard sense (although it is recursively specifiable), nor are the declarative and procedural facets entirely separate (in particular, the procedural consequence of a given expression may affect the subsequent context of use that determines what another expression designates). Nor are its consequences minor: we will we able to show, for example, that the traditional notion of evaluation is both confusing and confused, and must be separated into independent notions of *reference* and *simplification.* We will be able to show, in particular, that 1-LISP's evaluator de-references some expressions (such meta-syntactic terms as (QUOTE X), for example), and does not de-reference others (such as the numerals and T and NIL). We will argue instead for what we will call a *semantically rationalised* dialect, in which simplification and reference primitives are kept strictly distinct.

It is our view that semantical cleanliness is by far the most important pre-requisite to any conceivable treatment of reflection. However, as well as advocating *semantically rationalised* computational calculi, we will also espouse an aesthetic we call *category alignment*, by which we mean that there should be a strict category-category correspondence across the four major axes in terms of which a computation calculus is analysed: notation, abstract structure, declarative semantics, and procedural consequence (a mandate satisfied by no extant dialects). In particular, we will insist in the dialects we design that each *notational* class be parsed into a distinct *structural* class, that each structural class be treated

in a uniform way by the primitive processor, that each structural class serve as the normal-form designator of each semantical class, and so forth. This is an aesthetic with consequence: we will be able to show that the 1-LISP programmer must in certain situations resort to meta-syntactic machinery merely because 1-LISP fails to satisfy this mild requirement (in particular, 1-LISP lists, which are themselves a derivative class formed from some pairs and one atom, serve semantically to encode both function applications and enumerations). Though it does not have the same status as semantical hygiene, categorical elegance will also prove almost indispensible, especially from a practical point of view, in the drive towards reflection.

Once we have formulated these theoretical positions, we will be in a position to design 2-LISP. Like SCHEME and the λ-calculus, 2-LISP is a higher-order formalism: consequently, it is statically scoped, and treats the function position of an application as a standard extensional position. It is of course formulated in terms of our rationalised semantics, implying that a declarative semantics is formulated for all expressions prior to, and independent of, the specification of how they are treated by the primitive processor. Consequently, and unlike SCHEME, the 2-LISP processor is based on a regimen of *normalisation*, taking each expression into a normal-form designator of its referent, where the notion of *normal-form* is defined in part with reference to the semantic type of the symbol's designation, rather than (as in the case of the λ-calculus) in terms of the further (non-) applicability of a set of syntactic reduction rules. 2-LISP's normal-form designators are environment-independent and side-effect free; thus the concept of a *closure* can be reconstructed as a *normal-form function designator.* Since normalisation is a form of simplification, and is therefore *designation-preserving*, meta-structural expressions (terms that designate other terms in the language) are not de-referenced upon normalisation, as they are when evaluated. We will say that the 2-LISP processor is *semantically flat*, since it stays at a semantically fixed level (although explicit referencing and de-referencing primitives are also provided, to facilitate explicit shifts in level of designation).

3-LISP is straightforwardly defined as an extension of 2-LISP, with respect to an explicitly articulated procedural theory of 3-LISP embedded in 3-LISP structures. This embedded theory, called the *reflective model*, though superficially resembling a meta-circular interpreter (as a glance at the code, listed in S6-207, shows), is causally connected to the workings of the underlying calculus in critical and primitive ways. The reflective model is

similar in structure to the procedural fragment of the meta-theoretic characterisation of 2-LISP that we encoded in the λ-calculus: it is this incorporation into a system of a theory of its own operations that makes 3-LISP, like any possible reflective system, inherently theory relative. For example, whereas *environments* and *continuations* will up until this point have been theoretical posits, mentioned only in the meta-language, as a way of explaining LISP's behaviour, in 3-LISP such entities move from the semantical domain of the meta-language into the semantical domain of the object language, and environment and continuation designators emerge as part of the primitive behaviour of 3-LISP protocols.

More specifically, arbitrary 3-LISP *reflective* procedures can bind as arguments (designators of) the continuation and environment structure of the interpreter that would have been in effect at the moment the reflective procedure was called, had the machine been running all along in virtue of the explicit interpretation of the prior program, mediated by the reflective model. Furthermore, by constructing or modifying these designators, and resuming the process below, such a reflective procedure may arbitrarily control the processing of programs at the level beneath it. Because reflection may recurse arbitrarily, 3-LISP is most simply defined as an infinite tower of 3-LISP processes, each engendering the process immediately below, in virtue of running a copy of the reflective model. Under such an account, the use of reflective procedures amounts to running simple procedures at arbitrary levels in this reflective hierarchy. Both a straightforward implementation and a conceptual analysis are provided to demonstrate that such a machine is nevertheless finite.

The 3-LISP reflective levels are not unlike the levels in a typed logic or set theory, although of course each reflective level contains an omega-order untyped computational calculus essentially isomorphic to (the extensional portion of) 2-LISP. Reflective levels, in other words, are at once stronger and more encompassing than are the order levels of traditional systems. The locus of agency in each 3-LISP level, on the other hand, that distinguishes one computaional level from the next, is a notion without precedent in logical or mathematical traditions.

The architecture of 3-LISP allows us to unify three concepts of traditional programming languages that are typically independent (three concepts we will have explored separately in 1-LISP): a) the ability to support meta-circular interpreters, b) the

provision of explicit names for the primitive interpretive procedures (EVAL and APPLY in standard LISP dialects), and c) the inclusion of procedures that access the state of the implementation (usually provided, as part of a programming environment, for debugging purposes). We will show how all such behaviours can be defined within a pure version of 3-LISP (i.e., independent of implementation), since all aspects of the state of the 3-LISP interpretation process are available, with sufficient reflection, as objectified entities within the 3-LISP structural field.

The dissertation concludes by drawing back from the details of LISP development, and showing how the techniques employed in one particular case could be used in the construction of other reflective languages — reflective dialects of current formalisms, or other new systems built from the ground up. We will show, in particular, how our approach to reflection may be integrated with notions of data abstraction and message passing — two (related) concepts commanding considerable current attention, that might seem on the surface incompatible with the notion of a system-wide declarative semantics. Fortunately, we will be able to show that this early impression is false — that procedurally reflective *and semantically rationalised* variants on these types of languages could be readily constructed as well.

Besides the basic results on reflection, there are a variety of other lessons to be taken from our investigation, of which the integration of declarative import and procedural consequence in a unified and rationalised semantics is undoubtedly the most important. The rejection of evaluation, in favour of separate simplification and de-referencing protocols, is the major, but not the only, consequence of this revised semantical approach. The matter of category alignment, and the constant question of the proper use of meta-structural machinery, while of course not formal results, are nonetheless important permeating themes. Finally, the unification of a variety of practices that until now have be treated independently: macros, meta-circular interpreters, EVAL and APPLY, quotation, implementation-dependent debugging routines, and so forth, should convince the reader of one of our most important claims: procedural reflection is not a radically new idea; tentative steps in this direction have been taken in many areas of current practice. The present contribution — fully in the traditional spirit of rational reconstruction — is merely one of making explicit what we all already knew.

We conclude this brief introduction with three footnotes. First, given the flavour of the discussion so far, the reader may be tempted to conclude that the primary emphasis of this report is on procedural, rather than on representational, concerns (an impression that will only be reinforced by a quick glance through later chapters). This impression is in part illusory; as we will explain at a number of points, these topics are pursued in a procedural context because it is *simpler* than attempting to do so in a poorly understood representational or descriptive system. All of the substantive issues, however, have their immediate counterparts in the declarative aspects of reflection, especially when such declarative structures are integrated into a computational framework. Our investigation will always be carried on with the parallel declarative issues kept firmly in mind; the attribution of a declarative semantics to LISP s-expressions will also reveal our representational bias. As was mentioned in the preface, the decision to first explore reflection in a procedural context should be taken as methodological, rather than as substantive. Furthermore, it is towards a *unified* system that we are aiming; one of the morals under our present reconstruction is that the boundaries between these two types of calculus should ultimately be dismantled.

Secondly, as this last comment suggests, and as the unified treatment of semantics betrays, we consider it important to unify the theoretical vocabularies of the *declarative tradition* (logic, philosophy, and to a certain extent mathematics) with the *procedural tradition* (primarily computer science). The semantical approach we will adopt here is but a first step in that direction: as was mentioned in the first paragraph, a fully unified treatment remains an unattained goal. Nonetheless, considerable effort has been expended in the dissertation to present a single semantical and conceptual position that draws on the insights and techniques of both of these disciplines.

Third and finally, as the very first paragraph of this chapter suggests, the dissertation is offered as the first step in a general investigation into the construction of *generally* reflective computational calculi, to be based on more fully integrated theories of representation and computation. In spite of its reflective powers, and in spite of its declarative semantics, 3-LISP cannot properly be called fully reflective, since 3-LISP structures do not form a descriptive language (nor would any other procedurally reflective programming language that might be developed in the future, based on techniques set forth here, have any claim to the more general term). This is not because the 3-LISP structures

lack expressive power (although 3-LISP has no quantificational operators, implying that even if it were viewed as a descriptive language it would remain algebraic), but rather because all 3-LISP expressions are devoid of *assertional force.* There is, in brief, no way to say anything in such a formalism: we can set X to 3; we can test whether X is 3; but we cannot say *that* X is 3. Nevertheless, we contend that the insights won on the behalf of 3-LISP will ultimately prove useful in the development of more radical, generally reflective systems. In sum, we hope to convince the reader that, although it will be of some interest on its own, 3-LISP is only a corollary of the major theses adopted in its development.

1.b. The Concept of Reflection

In the present section we will look more carefully at what we mean by the term "reflection", in general and in the computational case; we will also specify what we would consider an acceptable theory of such a phenomenon. The structure of the solution we will eventually adopt will be presented only in section 1.e, after discussing in section 1.c the attendent model of computation on which it is based, and in section 1.d our conception of computational semantics. Before presenting any of that preparatory material, however, we do well to know where we are headed.

1.b.i. The Reflection and Representation Hypotheses

In the prologue we sketched with broad strokes some of the roles that reflection plays in general mental life. In order to focus the discussion, we will consider in more detail what we mean by the more restricted phrase "*computational* reflection". On one reading this term might refer to a successful computational model of general reflective thinking. For example, if you were able to formulate what human reflection comes to (presumably more precisely than we have been able to do), and were then able to construct a computational model embodying or exhibiting such behaviour, you would have some reason to claim that you had demonstrated computational reflection, in the sense of a *computational process that exhibited authentic reflective activity.*

Though we will work with this larger goal in mind, our use of the term will be more modest. In particular, we take no position on whether computational processes are able to "think" or "reason" *at all*; certainly it would seem that most of what we take computational systems to do is *attributed*, in a way that is radically different from the situation regarding our interpretations of the actions of other people. In particular, humans are first-class bearers of what we might call ***semantic originality***: they themselves are able to mean, without some observer having to attribute meaning to them. Computational processes, on the other hand, are at least not yet semantically original; to the extent they can be said to mean or refer at all, they do so *derivatively*, in virtue of some human finding that a convenient description (we duck the question as to whether it is a convenient *truth* or a convenient *fiction*).[2] For example, if, as you read this, you rationally and intentionally say

"*I am now reading section 1.b.i*", you succeed in referring to this section, without the aid of attendant observers. You do so *because we define the words that way:* reference and meaning and so on are paradigmatically and definitionally what people do. In other words your actions are the definitional locus of reference; the rest is hypothesis and falsifiable theory. On the other hand, if I inquire of my home computer as to the address of a friend's farm, and it tells me that it is on the west coast of Scotland, the computer has not referred to Scotland in any full-blooded sense: it hasn't a clue as to what or where Scotland is. Rather, it has typed out an address that it probably stored in an ASCII code, and *I* supply the reference relationship between that spelled word and the country in the British Isles.

The *reflection hypothesis* spelled out in the prologue, about how computational models of reflection might be constructed, embodied this cautionary stance: we said there that *in as much as a computational process can be constructed to reason at all,* it could be made to reason reflectively in a certain fashion. Thus our topic of computational reflection will be restricted to those computational processes that, for similar purposes, *we find it convenient to describe as reasoning reflectively.* In sum, we avoid completely the question of whether the "reflectiveness" embodied in our computational models is authentically borne, or derivately ascribed.

This is one major reduction in scope; we immediately adopt another. Again, in the prologue, we spoke of reflection as if it encompassed contemplative consideration both of one's world and of one's self. We will discuss the relationship between reflection and self-reference in more detail below, but we should admit at the outset that the focus of our investigation will be almost entirely on the "selfish" part of reflection: on what it is to construct computational systems able to deal *with their own ingredient structures and operations* as explicit subject matters. The reasons for this constraint on our investigation are worth spelling out. It might seem as if this restriction arises for simple reasons, such as that this is an easier and better-constrained subject matter (since after all we are in no position to postulate models of thinking about external worlds). However in fact this restriction in scope arises for deeper reasons, again having to do with the reflection hypothesis. First, we will consider *internal* or *interior* processes able to reflect on *interior* structures, which is the only world that those internal processes conceivably can have any access to. For example, we will construct a particular kind of LISP processor (interpreter),

and LISP processors have no access to anything except fields of LISP s-expressions. On the other hand LISP processors are crucially *interior* processes (in a sense that will be made clear in the next section): they do not interact with the world directly, but rather, in virtue of running programs, engender more complex processes that interact with the world.

This "interior" sense of language processors interacts crucially with the reflection hypothesis, especially in conjunction with the representation hypothesis. Not only can we restrict to our attention to ingredient processes "reasoning about" (computing over, whatever) internal computational structures, we can restrict our attention to processes *that shift their (extensional) attention to meta-structural terms.* For consider: if it turns out that I am a computational system, consisting of an ingredient process P manipulating formal representations of my knowledge of the world, then when I think, say, about Virginia Falls in northern Canada, my ingredient processor P is manipulating *representations* that are about Virginia Falls. Suppose, then, that I back off a step and comment to myself that whenever I should be writing another sentence I have a tendency instead to think about Virginia Falls. What do we suppose that my processor P is doing now? Presumably ("presumably", at least, according to the knowledge representation hypothesis, which, it is important to reiterate, we are under no compulsion to believe) my processor P is now manipulating representations *of* my representations of Virginia Falls. In other words, *because we are focussed on the behaviour of interior processes,* not on compositionally constituted processes, *our exclusive focus on self-referential aspects of those processes is all we can do* (given our two governing hypotheses) *to uncover the structure of constituted, genuine reflective thought.*

We can put this same point another way. The reflection hypothesis does not state that, in the circumstance just described, P will *reflect* on the knowledge structures representing Virginia Falls (in some weird and wondrous way) — this would be an unhappy proposal, since it would not offer any hope of an *explanation* of reflection. Reflective behaviour — the subject matter to be explained — should presumably not occur as a phenomenon in the explanation. Rather, the reflection hypothesis is at once much stronger and more tractable (although perhaps for that very reason less plausible): it posits, as an explanation of the mechanism of reflection, that the interior process compute over *a different kind of symbol.* The most important feature of the reflection hypothesis, in other words, is its tacit assumption that the computation engendering reflective reasoning,

although it may be over a different kind of structure, is nonetheless *similar in kind* to the sorts of computation that regularly proceed over normal structures.

In sum, it is our methodological allegience to the knowledge representation hypothesis that underwrites our self-referential stance. Though we will not mention this meta-theoretic position further, it is crucial that it be understood, for it is only because of it that we have any right to call our inquiry a study of *reflection*, rather than a (presumably less interesting) study of *computational self-reference.*

1.b.ii. Reflection in Computational Formalisms

With these preliminaries set straight, we may turn, then, to the question of what it would be to make a computational process reflective in this sense.

At its heart, the problem derives from the fact that in traditional computational formalisms the behaviour and state of the interpretation process are not accessible to the reasoning procedures: the interpreter forms part of the tacit background in terms of which the reasoning processes work. Thus, in the majority of programming languages, and in all representation languages, only the un-interpreted data structures are within the reach of a program. A few languages, such as LISP and SNOBOL, extend this basic provision by allowing program structures to be examined, constructed, and manipulated as first class entities. What has never been provided is a high level language in which the process that interprets those programs is also visible and subject to modification and scrutiny. Therefore such matters as whether the interpreter is using a depth-first control strategy, or whether free variables are dynamically scoped, or how long the current problem has been under investigation, or what caused the interpreter to start up the current procedure, remain by and large outside the realm of reference of the standard representational structures. One way in which this limitation is partially overcome in some programming languages is to allow procedures access to the structures of the *implementation* (examples: MDL, INTERLISP, etc.[3]), although such a solution is inelegant in the extreme, defeats portability and coherence, lacks generality, and in general exhibits a variety of mis-features we will examine in due course. In more representational or declarative contexts no such mechanism has been demonstrated, although a need for some sort of reflective power has appeared in a variety of contexts (such as for over-riding defaults, gracefully handling contradictions, etc.).

A striking example comes up in problem-solving: the issue is one of enabling simple declarative statements to be made about how the deduction operation should proceed. For example, it is sometimes suggested that a *default* should be implemented by a deductive regime that accepts inferences of the following non-monotonic variety:

$$\frac{\neg \vdash \neg P}{P} \qquad \text{(S1-1)}$$

Though it isn't difficult to build a problem solver that *embodies* some such behaviour (at least on some computable reading of "not provable"), one typically doesn't want such a rule to be obeyed indiscriminately, independent of context or domain. There are, in other words, usually constraints on when such inferences are appropriate, having to do with, say, how crucially the problem needs a reliable answer, or with whether other less heuristic approaches have been tried first. What we are after is a way to write down specific instances of something like S1-1 that refer explicitly both to the subject domain and to the state of the deductive apparatus, and that, *in virtue of being written down*, lead that inference mechanism to behave in the way described.

Particular examples are easy to imagine. Consider, for instance, a computational process designed to repair electronic circuits. One can imagine that it would be useful to have inference rules of the following sort: *"unless you have been told that the power supply is broken, you should assume that it works"*, or, *"you should make checking capacitors your first priority, since they are more likely to break down than are resistors"*. Furthermore, we would like ensure that such rules could be modularly and flexibly added and removed from the system, without each time requiring surgery on the inner constitution of the inference engine. Though we are skirting close to the edge of an infinite regress, it is clear that something like this kind of protocol is a natural part of normal human conversation. From an *intuitive* point of view it doesn't seem unreasonable to say, *"By the way, if you ever want to assume P, it would be sufficient to establish that you cannot prove its negation."*; the question is whether we can make *formal* sense out of this intuition.

It is clear that the problem is not so much one of *what* to say, but of *how* to say it (say, to some kind of theorem-prover) in a way that doesn't lead to an infinite regress, and that genuinely affects its behaviour. All sorts of technical questions arise. It is not obvious, for example, what language to use, or even to whom such a statement should be directed. Suppose, for example, that we were given a monotonic natural-deduction based theorem

prover for first order logic. Could we give it S1-1 as an implication? Certainly not; S1-1, at least in the form given above, is not even a well-formed sentence. There are various ways we could *encode* it as a sentence — one way would be to use set theory, and to talk explicitly about the set of sentences derivable from other sentences, and then to say that if the sentence "¬P" is not in a certain set, then "P" is. However, although such a sentence might contribute to a *model* of the kind of inference procedure we desire, it wouldn't *make the current inference mechanism behave non-monotonically.* To do this would not be to construct a non-monotonic reasoning system, but rather to build a monotonic one prepared to reason about a non-monotonic one. While such a formulation might be of interest in the specification of the constraints a reasoning system must honour (a kind of "competence theory" for non-monotonic reasoning[4]), it doesn't help us, at least on the face of it, with the question of how a system using defaults might actually be deployed. Another option would be to build a non-monotonic inference engine from scratch, using expressions like S1-1 to *constrain* its behaviour, like the abstract specifications of a program. But this would solve the problem by avoiding it — the whole question was how to use such comments on the reasoning procedure coherently within the structures of the problem-specific application.

Yet another possibility — and one we will focus on for a moment — would be to design a more complex inference mechanism to react appropriately not only to sentences in the standard object language, but to meta-theoretic expressions of the form S1-1. Although no system claiming to be of just this sort has been demonstrated, such a program is readily imagineable, and various dialects of PROLOG — perhaps most clearly the IC-PROLOG of Imperial College[5] — are best viewed in this light. The problem with such solutions, however, is their excessive rigidity and inelegance, coupled with the fact that they don't really solve the problem in any case. What a PROLOG user is given is not a unified or reflective system, but a pair of two largely *independent* formal systems: a basic declarative language in which facts about the world are expressed, and a procedural language, in which the behaviour of the inference process is controlled. Although the elements of the two languages are mixed in a PROLOG program, they are best understood as separate aspects. One set (the clause and implication and predicate structure, the identity of the variables, and so forth) constitutes the *declarative* language, with the standard semantics of first-order logic. Another (the sequential ordering of the sentences and of the predicates in the premise, the "consumer" and "producer" annotations on the variables, the "cut" operator,

and so forth) constitute the *procedural* language. Of course the flow of control is affected by the declarative aspects, but this is just like saying that the flow of control of an ALGOL program is affected by the data structures. Thus the claim that to use PROLOG is to "program in logic" is rather misleading: instead one essentially writes programs in a new (and, as it happens, rather limited) control language, using an encoding of first-order logic as the declarative representation language. Of course this is a dual system with a striking fact about its procedural component: all conclusions that can be reached are guaranteed to be valid implications of prior structures in the representational field. However, as was mentioned above, this kind of dual-calculus approach seems ultimately rather baroque, and is certainly not conducive to the kind of reflective abilities we are after. It would surely be far more elegant to be able to say, *in the same language as the target world is described,* whatever it was salient to say about how the inference process was to proceed. For example, to continue with the PROLOG example, one would like to say both FATHER(BENJAMIN,CHARLES) *and* CUT(CLAUSE-13) or DATA-CONSUMER(VARIABLE-4), in the same language and subject to the same semantical treatment. The increase in elegance, expressive power, and clarity of semantics that would result are too obvious to belabour: just a moment's thought leads to one realise that one a single semantical analysis would be necessary (rather than two); the reflective capabilities could recurse without limit (in PROLOG and other dual-calculus sytems there is only one level); a meta-theoretic description of the system would have to describe only one formal language, not two; descriptions of the inference mechanism would be immediately available, rather than having to be extracted from procedural code; and so forth.

The ability to pass coherently between two situations: in the reflective case to have the structures that normally control the interpretation process be fully and explicitly visible to (and manipulable by) the reasoning process, and in the other to allow the reasoning process to sink into them, so that they may take their natural effect as part of the tacit background in which the reasoning process works — this ability is a particular form of reflection we will call ***procedural reflection*** ("procedural" because we are not yet requiring that those structures at the same time *describe* the reasoning behaviours they engender: that is a larger task). Though ultimately limited, in the sense that a procedurally reflective calculus is by no means a fully reflective one, even this more modest notion is on its own a considerable subject of inquiry.

1.b.iii. Six General Properties of Reflection

Given the foregoing sketch of what our task is, it is appropriate to ask, before plunging into details, whether we have any sense in advance of what form our solution might take. Six properties of reflective systems can be identified straight away — features that we will expect our ultimate solutions to exhibit, however they end up being structured or explained.

First, the notion is one of self-reference, of a *causally-connected* kind, stronger than the notions explored by mathematicians and philosophers over much of the last century. What we need is a theory of the causal powers required in order that a system's possession of self-descriptive and self-modelling abilities will *actually matter* to it — a requirement of substance since full-blooded, actual behaviour is our ultimate subject matter, not simply the mathematical characterisation of formal relationships. In dealing with computational processes, we are dealing with artefacts *behaviourally* defined, unlike systems of logic which are *functionally* defined abstractions that in no way behave or participate with us in the temporal dimension. Although any abstract machine of Turing power can provably model any other — including itself — there can be no sense in which such self-modelling is even *noticed* by the underlying machine (even if we could posit an *animus ex machina* to do the noticing). If, on the other hand, we aim to build a computational system of substantial reflective powers, we will have to build something that is affected by its ability to "think about itself". This holds no matter how accurate the self-descriptive model may be; you simply cannot afford simply to reason about yourself as disinterestedly and inconsequentially as if you were someone else.

Similar requirements of *causal connection* hold of human reflection. Suppose, for example, that after taking a spill into a river I analyse my canoeing skills and develop an account of how I would do better to lean downstream when exiting an eddy. Coming to this realisation is useful just in so far as it enables me to improve; if I merely smile in vacant pleasure at an *image* of an improved me, but then repeat my ignominious performance — *if, in other words, my reflective contemplations have no effect on my subsequent behaviour* — then my reflection will have been worthless. The move has to be made, in other words, from description to reality. In addition, just as the *result* of reflecting has to affect *future* non-reflective behaviour, so does *prior* non-reflective

behaviour have to be accessible to reflective contemplation; one must also be able to move from reality to description. It would have been equally futile if, when I paused initially to reflect on the cause of my dunking, I had been unable to remember what I had been doing just before I capsized.

In sum, the relationship between reflective and non-reflective behaviour must be of a form such that both information and effect can pass back and forth between them. These requirements will impinge on the technical details of reflective calculi: we will have to strive to provide sufficient connection between reflective and non-reflective behaviour so that the right causal powers can be transferred across the boundary, without falling into the opposite difficulty of making them so interconnected that confusion results. (An example is the issue of providing continuation structures to encode control flow: we will provide *separate* continuation structures for each reflective level, to avoid unwanted interactions, but we will also have to provide a way in which a designator of the lower level continuation can be bound in the environment of the higher one, so that a reflective program can straightforwardly refer to the continuation of the process below it.) Furthermore, the interactions can become rather complex. Suppose, to take another example, that you decide at some point in your life that whenever some type of situation arises (say, when you start behaving inappropriately in some fashion), that you will pause to calm yourself down, and to review what has happened in the past when you have let your basic tendencies proceed unchecked. The dispassionate fellow that you must now become is one that embodies a decision *at some future point to reflect.* Somehow, without acting in a self-conscious way from now until such a circumstance arises, you have to make it true that when the situation *does* arise, you will have left yourself in a state that will cause the appropriate reflection to happen. Similarly, in our technical formalisms, we will have to provide the ability to drop down from a reflected state to a non-reflected one, having left the base level system in such a state that when certain situations occur the system will automatically reflect, and thereby obtain access to the reasons that were marshalled in support of the original decision.

Second, reflection has something — although just what remains to be seen — to do with self-*knowledge*, as well as with self-*reference*, and knowledge, as has often been remarked, is inherently *theory-relative.* Just as one cannot interpret the world except by using the concepts and categories of a theory, one cannot reflect on one's self except with reference to a theory of oneself. Furthermore, as is the case in any theoretical endeavour,

the phenomena under consideration under-determine the theory that accounts for them, even when all the data are to be accounted for. In the more common case, when only parts of the phenomenal field are to be treated by the theory, an even wider set of alternative theories emerge as possibilities. In other words, *when you reflect on your own behaviour, you must inevitably do so in a somewhat arbitrary theory-relative way.*

One of the mandates we will set for any reflective calculus is that it be provided, represented in its own internal language, with a complete (in some appropriate sense) theory of how it is formed and of how it works. Theoretical entities may be posited by this account that facilitate an explanation of behaviour, even though those entities cannot be claimed to have a theory-independent ontological existence in the behaviour being explained. For example, 3-LISP will be provided with a "theory", in 3-LISP, of 3-LISP (reminiscent of the meta-circular interpreters demonstrated in McCarthy's original report[6] and in the reports of Sussman and Steele,[7] but causally connected in novel ways). In providing this primitively supported reflective model, we will adopt a standard account, in which many common notions of LISP (such as the notion of an *environment* just mentioned, and a parallel notion of a *continuation*) play a central role, even though they are not first-class objects *of the language* in any direct sense. It is impossible *in a non-reflective LISP* to define a predicate true only of environments, since environments as such don't exist in non-reflective LISP's. However, once we endow our particular dialect with reflective powers, the notion of an environment will be crucial, and environments will be passed around as first-class objects.

There are other possible LISP theories, some of which differ radically from the one we have chosen. It is possible, for example, to replace the notion of environment altogether (note that the λ-calculus is explained without any such device). But the point is that in building a reflective model based on this alternative theory, other objects would probably be posited instead: in order to reflect you have to use *some* theory and its associated theoretical entities.

The third general point about reflection regards its name: we deliberately use the term "reflective", as opposed to "reflexive", since there are various senses (other recent research reports not withstanding[8]) in which no computational process, in any sense that this author can understand, can succeed narcissistically in thinking *about the fact that it is*

at that very instant thinking about itself thinking about itself thinking ... — and so on and so on, like a transparent eye in a room full of mirrors. The kind of reflecting we will consider — the kind that we will be able technically to define, implement, and control — requires that in the act of reflecting the process "take a step back", in order to allow the interpreted process to consider what it was just up to: to bring into view formal symbols which describe its state "just a moment earlier". From the fact of having a name for itself it does not automatically acquire the ability to *focus on its current instantaneous self*, for in the process of "stepping back" or reflecting, the "mind's eye" moves out of its own view, being replaced by an (albeit possibly complete) account of itself. (Though this description is surely more suggestive than incisive, much of the technical work to be presented will allow us to make it precise.)

The fourth comment is that, in virtue of reflecting, a process can always obtain a finer-grained control over its behaviour than would otherwise be possible. What was previously an inexorable stepping from one state to the next is opened up so that each move can be analysed, countered, and so forth. In other words we will see in great detail how reflective powers in fact provide for a more subtle and more catholic — if less efficient — way of reacting to a world. The requirement here is as usual for what was previously implicit to be made explicit, albeit in a controlled and useful way, without violating the ultimate truth that not everything can be made explicit in a finite mechanism. This ability enables a system designer to satisfy what might be taken as incompatible demands: the provision of a small and elegant kernel calculus, with crisp definition and strict behaviour, and at the same time provide (through reflection) the user with the ability to modify or adjust the behaviour of this kernel in peculiar or extenuating circumstances. Thus simplicity and flexibility can be achieved together.

This leads us to the fifth general comment, which is that the ability to reflect never provides a complete separation, or an utterly objective vantage point from which to view either oneself or the world. No matter now reflective any given person may be, it is a truism that there is ultimately no escape from being the person in question. Though we will generally downplay any connection between our formal work and human abilities, we can perhaps allow that the kind of reflection we are modelling is closer to what is known as *detachment* or *awareness* than to a strict kind of self-objectivity (this is why we are systematically and intentionally imprecise about whether *reflection* is focused on the self or

on the world). The environment example just mentioned provides an illustration of this in a computational setting. As we will see in detail, the environment in which are bound the symbols that a program is using is, at any level, merely part of the embedding background in which the program is running. The program operates within that background, dependent on it but — in the normal (non-reflective) course of events — unable to access it explicitly. The operation of reflecting makes explicit what was just implicit: it renders visible what was tacit. In doing so, however, a new background fills in to support the reflection. Again, the same is true of human reflection: you and I can interrupt our conversation in order to sort out the definition of a contentious term, but — as has often been remarked — we do so using other terms. Since language is our inherent medium, we cannot step out of it to view it from a completely independent vantage point. Similarly, while the systems we build will at any point be able to back up and *mention* what was previously *used*, in doing so more structures will come into implicit use. This lesson, of course, has been a major one in philosophy at least since Peirce; certainly Quine's lesson of Neurath's boat holds as true for the systems we design as it does for us designers.[9]

Sixth and finally, the ability to reflect is something that must be built into the heart or kernel of a calculus. There are theoretically demonstrable reasons why it is not something which can be "programmed up" as an addition to a calculus (although one of course can *implement* a reflective machine in a non-reflective one: the difference between these two must always be kept in mind). The reason for this claim is that, as discussed in the first comment, *being reflective is a stronger requirement on a calculus than simply being able to model the calculus in the calculus*, something any machine of Turing power is capable of doing (this is the "making it matter" that was alluded to above). This will be demonstrated in detail; the crucial difference, as suggested above, comes in connecting the self-model to the basic interpretation functions in a causal way, so that (for example and very roughly) when a process "decides to assume something", it in fact assumes it, rather than simply constructing a model or self-description or hypothesis that *says* that it is in fact assuming it. As well as "backing up" in order to reflect on its thoughts, in other words, the process needs to be able to "drop back down again", to consider the world directly, in accord with the consequences of those reflections. Both parts of this involve a causal connection between the explicit programs and the basic workings of the abstract machine, and such connections cannot be "programmed into" a calculus that does not support them

primitively.

1.b.iv. Reflection and Self-Reference

At the beginning of this section we said that our investigation of reflection in general would primarily concern itself, because of the knowledge representation hypothesis, with the *self-referential* aspects of reflective behaviour. There has been in the last century no lack of investigation into self-referential expressions in formal systems, especially since it has been exactly in these areas where the major results on paradox, incompleteness, undecidability, and so forth, have arisen. We should therefore compare our enterprise with these theoretical precursors.

Two facets of the computational situation show how very different our concerns will be from these more traditional studies. First, although we do not formalise this, there is no doubt in our work that we consider the locus of *referring* to be *an entire process*, not a particular expression or structure. Even though we will posit declarative semantics for individual expressions, we will also make evident the fact that the designation of any given expression is a function not only of the expression itself, but also of the state of the processor *at the point of use* of that expression. And of course it is the processor that uses the symbol; the symbol does not use itself. To the extent that we want our system to be self-referential, then, we want *the process as a whole* to be able to refer, to first approximation, *to its whole self*, although in fact this usually reduces to a question of it refering to some of its own ingredient structure.

We do not typically want specific structures themselves to be self-designating, exactly to avoid many of the intractable (if not inscrutable) problems that arise in such cases. It will be perfectly possible to construct apparently self-designating expressions (at least up to type-equivalence: token self-reference is more difficult). But by and large the system of levels we will adopt will exclude such local self-reference, practically if not formally, from our consideration. Truly self-referential expressions, such as *This sentence is six words long*, are unarguably odd, and certain instances of them, such as the clichéd *This sentence is false*, are undeniably problematic (strictly, of course, the sentence "This sentence is six words long" *contains* a self-reference, but is not itself self-referential; however we could use instead the composite term "*This five word noun phrase*"). None of these truths impinge particularly on our quite different concerns.

The second major comment is this: in traditional formal systems, the *actual reference* relationship between any given expression and its referent (be that referent itself or a distal object) is mediated by the externally attributed semantical interpretation function. The sentence *"This sentence is six words long"* doesn't actually *refer*, in any causal full-blooded sense, to anything; rather, we English speakers *take it* to refer to itself. The causal reference relationship between that sentence as sign, and that sentence as significant, flows through us.

As we said in the previous section about causal connection, in constructing reflective computational systems it is crucial that we *not* defer causal mediation through an external observer. Reflection in a computational system *has to be causally connected*, even if the *semantical* understanding of that causal connection is externally attributed. For example, in 3-LISP there is a primitive relationship that holds between a certain kind of symbol, called a *handle* (a canonical form of meta-descriptive rigid designator) and another symbol that, informally, each handle designates. Suppose that H_1 is some handle, and that S_1 is some structure that H_1 refers to; strictly speaking the relationship between H_1 and S_1 is an internal relationship, that we, as external semantical attributors, take to be a reference relationship. Until we can construct computational systems that are what we called semantically original, the *semantical* import of that relationship remains external. But the *causal* relationship between H_1 and S_1 *must be internal*: otherwise there would be no way for the internal computational processes to treat that relationship in any way that mattered.

We can put this a little more formally, which may make it clearer. Suppose that Φ is the externally attributed semantical interpretation function, and that Ξ is the primitive function that relates handles to the structures we call their referents. Thus we have, to use the prior example, $[\Phi(H_1) = S_1]$, as well as $[\Xi(H_1) = S_1]$. More generally, we know that:

$$\forall H,S\ [[\text{HANDLE}(H)] \wedge [\Xi(H) = S]] \supset [\Phi(H) = S]] \qquad \text{(S1-2)}$$

However this equation, though in some sense strictly true, in no way reveals the *structure* of the relationship between Φ and Ξ; it merely states their extensional equivalence. More revealing of the fact that we take the relationship between handles and referents to be a reference relationship, if we are allowed to reify relationships, is the following:

$$\Phi(\Xi) = \Phi \qquad \text{(S1-3)}$$

or, rather, since not all symbols are handles, as:

$$\Phi(\Xi) \subset \Phi \qquad \text{(S1-4)}$$

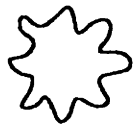
The requirement that reflection *matter*, to summarise, is a crucial facet of computational reflection — one without precedent in pre-computational formal systems. What is striking is that the mattering cannot be derived from the semantics, since it would appear that mattering — real causal connections — are a *precursor* to semantical originality, not something that can *follow* the semantical relationships. Put another way, in the inchoately semantical computational systems we are presently able to build, the reference relationships between internal meta-level symbols and their internal referents (these are the semantical relationships that are crucial in reflective considerations) may have to be causal in *two distinct ways*: once mediated by us who attribute semantics to those symbols in the first place, and once internally so that the appropriate causal behaviour, to which we attribute semantics, can be engendered. On that day when we succeed in constructing semantically original mechanisms, those two presently independent causal connections may merge; until then we will have to content ourselves with *causally original* but *semantically derivative* systems. The reflective dialects we will examine will all be of this form.

1.c. A Process Reduction Model of Computation

We need to sketch the model of computation on which our analysis will depend. We take ***processes*** as our fundamental subject matter; though we will not define the concept precisely, we may assume that a process consists approximately of a connected or coherent set of events through time. The reification of processes as objects in their own right — composite and causally engendered — is a distinctive, although not distinguishing, mark of computer science. Processes are inherently temporal, but not otherwise physical: they do not have spatial extent, although they *must* have temporal extent. Whether there are more abstract dimensions in which in is appropriate to locate a process is a question we will side-step; since this entire characterisation is by way of background for another discussion, we will rely more on example, and on the uses to which we put these objects, than on explicit formulation.

We will often depict processes as rough-edged circles or balls, as in the following diagram. The icon is intended to signify what we will call the ***boundary*** or ***surface*** of the process, which is the interface between the process and the world in which it exists (we presume that in virtue of objectifying processes we carve them out of a world in which they can then be said to be embedded). Thus the set of events that collectively form a coherent process in a given world will all be events on the surface of this abstract object. In any given circumstance this set of events could presumably be more or less specifically described: we might simply say that the process had certain gross input/output behaviour ("input" and "output" would have to be defined as surface perturbations of a certain class: this is an interesting but non-trivial problem), or we might account in fine detail for every nuance of the process's behaviour, including the exact temporal relationships between one event and the next, and so forth.

(S1-6)

PROCESS P 

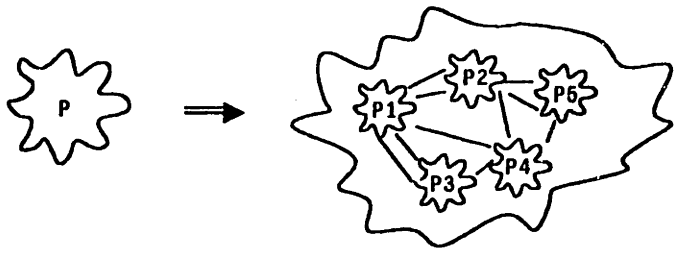
It is crucial to distinguish more and less fine-grained accounts of the surface of a process, on the one hand, from compositional accounts of its interior, on the other. That a process has an interior is again a striking assumption throughout computer science: the role

of *interpreters* (what we will call ***processors***) is a striking example. Suppose for instance that you interact with a so-called LISP-based editor. It is standard to assume that the LISP interpreter is an ***ingredient process*** within the process with which you interact: it in fact is the locus of *anima* or *agency* inside your editor process that supplies the temporal action in the editor. On the other hand that process never appears as the surface of the editor: no editor interaction is directly an interaction with the LISP processor. Rather, the LISP processor, in conjunction with some appropriate LISP *program*, together engender the behavioural surface with which you interact.

There are a variety of architectures — or classes of architecture — that computer science has studied; we will briefly mention just two, but will focus throughout the dissertation on just one of these. *Every* computational process (we will examine in a moment which processes we are disposed to call computational) has within it at least one other process: this supplies the animate agency of the overall constituted process. It is for this reason that we call this model a "process reduction" model of computation, since at each stage of ***computational reduction*** a given process is reduced in terms of constituent symbols and other processes. There may be more than one internal process (in what are known as *parallel* or *conconcurrent* processes), or there may be just a single one (known as *serial* processes). Reductions of processes which do not posit an interior process as the source of the agency we will consider outside the proper realm of computer science, although of course *some* such reduction must at some point be accounted for if the engendered process is ever to be realised. However this kind of reduction from process to, say, behaviour of physical mechanism, is more the role of physics or electronics than computer science *per se*. What is critical is that at some stage in a series of computational reductions this leap from the domain or processes to the domain of mechanisms be taken, as for example in the explaining how the behaviour of a set of logic circuits constitutes a processor (interpreter) for the micro-code of a given computer. Given this one account of what we may call the ***realisation*** of a computational process, then an entire hierarchy of processes above it may obtain indirect realisation. If, for example, that micro-code processor interprets a set of instructions that are the program for a macro-machine, then a macro-processor may thereby exist. Similarly, that macro-machine may interpret a machine language program that implements SNOBOL: thus by two stages of *composition* (the inverse of reduction) a SNOBOL processor is also realised.

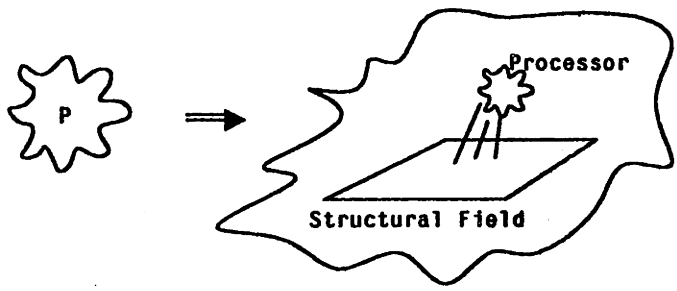
In order to make this talk of processors and so forth a little clearer, we show in the following diagrams two quite different forms of computational reduction: what we will call a ***communicative reduction*** and an ***interpretive reduction***. The arrow is intended to mean "reduces to"; thus in S1-6 we imply that process P reduces to a set of five interior processes. What it is for processes to communicate we will not say: the assumption is merely that these five ingredient processes interact in some fashion, so that taken as a composite unity their total behaviour is (i.e., can be interpreted as) the behaviour of the constituted process. Responsibility for the surface of the total process P is presumably shared in some way amongst the five ingredients. Examples of this sort of reduction may be found at any level of the computational spectrum, from metaphors of disk-controllers communicating with bus mediators communicating with central processors, to the message-passing metaphors in such AI languages as ACT1 and SMALLTALK, and so forth.[10]

(S1-6)



Communicative reductions will receive only passing mention in this dissertation; we discuss them here only in order to admit that the model of reflection that we will propose is not (at at least at present) sufficiently general to encompass them. We will focus instead on the far more common model that we call an *interpretive reduction*, pictured in the following diagram. In such cases the overall process is composed of what we will call a ***processor*** and a ***structural field***. The first ingredient is the locus of active agency: it is what is typically called an "interpreter", although we avoid that term because of its confusion with notions of interpretation from the declarative tradition (we will have much more to say about this confusion in chapter 3). The second is the program or data structures (or both): it is often called a set of *symbols*, although that term is so semantically loaded that we will avoid it for the time being.

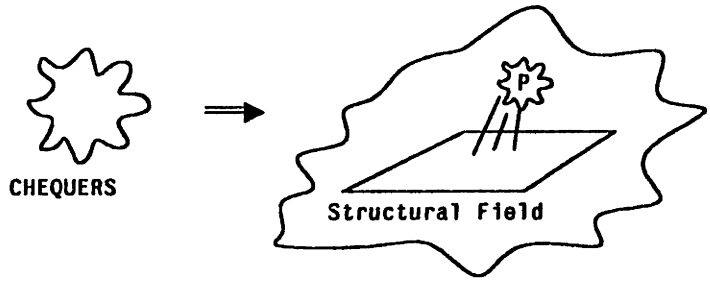
(S1-7)



All of the standard interpreted languages are examples of this second kind of reduction, of which LISP is as good an instance as any. The structural field of LISP consists of what are known as *s-expressions:* a combination of pairs (binary graph elements of a certain form), atoms, numerals, and so forth.

We intend the interpretive model to underwrite both language design and the construction of particular programs. For example, we can characterise FORTRAN in these terms: we will posit a FORTRAN processor that computes over (examines, manipulates, constructs, reacts to, and so forth) elements of the FORTRAN structural field, which includes primarily an ordered sequence of FORTRAN instructions, FORMAT statements, etc. Suppose that you set out to build a FORTRAN program to manage your financial affairs: what you would do is specify a set of FORTRAN data structures and a process to interact with them. We might call those data structures — the tables that list current balance, recent deposits, interest rate, and so on — the structural field of process CHEQUERS that you are building. The program that you want to interact with this data base we will simply call P. Thus the first reduction of CHEQUERS would be pictured in our model as follows:

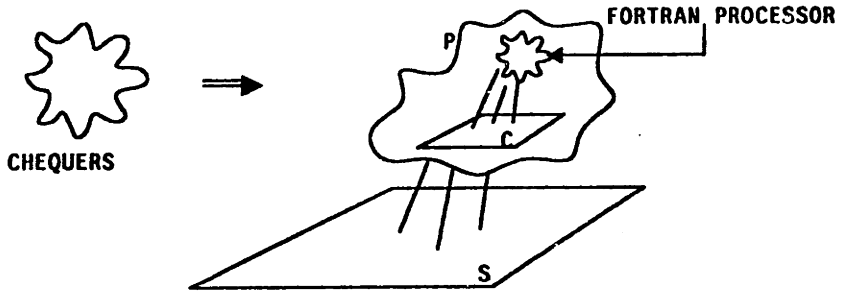
(S1-8)



We have said, however, that P is specified by a FORTRAN program (P is not itself a program, because P is a process, and programs are static, requiring interpretation by a processor in order to engender behaviour). Thus P can itself be understood in terms of a reduction in

terms of the program C ("C" for "code"), which when processed by the FORTRAN processor yields process P. Thus we have a double reduction of the following sort:

(S1-9)



There are a host of questions that would have to be answered before we could make this precise (before, for example, we could construct an adequate mathematical treatment of these intuitions). For example, the data structures in the foregoing example are themselves have to be implemented in FORTRAN as well. However to fill out the model just a little, we can suggest how we might, in these terms, define a variety of commonplace terms of art of computer science.

First, by the computer science term ***interpreter*** (again, we use instead "processor") we refer to a process *that is the interior process in an interpretive reduction of another interior process.* For example, the process P in the check-book example was not an interpreter, because it was the ingredient process only singly: the process thereby constituted, which we called CHEQUERS, was not *itself* an interior process. Hence P fails to be an interpreter. The reason that we call the process that interprets LISP programs an interpreter is because LISP programs are structural field arrangements that engender other interior processes that work over data structures so as to yield yet other processes.

Second, by a ***compilation*** we refer to the transformation or translation of a structural field arrangement S_1 to another structural field arrangement S_2, so that the surface of the process that would be yielded by the processing of S_1 by some processor P_1 is equivalent (modulo some appropriate equivalence metric) to the processing of S_2 by some processor P_2. For example, we spoke above about a FORTRAN processor, but of course such a processor is rarely if ever realised; rather, FORTRAN programs are typically compiled into some machine language. Suppose we consider the compiler that compiles FORTRAN into the machine language of the IBM 360. Then the compilation of some FORTRAN program C_F into an IBM

360 machine language porgram C_{360} would be correct just in case the surface of the process that would result from the processing of C_F by the (hypothetical) FORTRAN processor would be equivalent to the process that will actually result by the processing of C_{360} by the basic IBM 360 machine language processor. Thus compilation is relative to two reductions, and is mandated only to ensure surface-surface equivalence.

Third, by ***implementation*** we typically refer to two kinds of construction. To implement a *process* simply means to construct a structural field arrangement S for some processor P so that the surface of the process that results from the interpretation of S by P yields the desired behaviour. More interesting is to implement a *language* (by a computational ***language*** we mean an architecture of a structural field and a behaviourally specified processor that interprets arrangements of such a field). In its most general form, one implements a language by providing a process P that can be reduced to the structural field and interior processor of the language being implemented. In other words if I implement LISP, all I am required to do is to provide a process that *behaviourally* appears to be a constituted process consisting of the LISP structural field and the interior LISP processor. Thus I am completely free of any actual commitment as to the reality, if any, of the implemented field.

Typically, one language is implemented in another by constructing some arrangement or set of protocols on the data structures of the implementing language to encode the structural field of the implemented language, and by constructing a program in the implementing language that, when processed by the implementing language's processor, will yield a process whose surface can be taken as a processor for the interpreted language, with respect to that encoding of the implemented language's structural field. (By a ***program*** we refer to a structural field arrangement *within an interior processor* — i.e., to the inner structural field of a double reduction — since programs are structures that are interpreted to yield processes that in turn interact with another structural field (the data structures) so as to engender a whole constituted behaviour.)

Finally, we can imagine how this model could be used in cognitive theorising. A ***weak*** computational model of some mental phenomenon would be a computational process that was claimed to be superficially equivalent to some mental behaviour. Note that surface equivalence of this sort can be arbitrarily fine-grained; just because a given computational model predicts the most minute temporal nuances revealed by click-stop

experiments and so forth does not imply that anything other than surface equivalence has been achieved. In contrast, a ***strong*** computational model would posit not only surface but interior architectural structure. Thus for example Fodor's recent claim of mental modularity[11] is a coarse-grained but strong claim: he suggests that the dominant or overarching computational reduction of the mental is closer to a communicative than to an interpretive reduction.

This has been the briefest of sketches of a substantial subject. Ultimately, it should be formalised into a generally applicable and mathematically rigourous account, but in this dissertation we will merely use its basic structure to organise our particular analyses. However there are three properties of all structural fields that are important for us to make clear, for the present investigation. First, over every structural field there must be defined a locality metric or measure, since *the interaction of a processor with a structural field is always constrained to be locally continuous.* Informally, we can think of the processor looking at the structural field with a pencil-beam flashlight, able to see and react only to what is currently illuminated (more formally, the behaviour of the processor must always be a function only of its internal state plus the current single structural field element under investigation). Why it is that the well-known joke about a COME-FROM statement in FORTRAN is funny, for example, can be explained only because this local accessibility constraint is violated (otherwise it would be a perfectly well-defined construct). Note as well that in logic, the λ-calculus, and so forth, no such locality considerations come into play. In addition, the measure space yielded by this locality metric need not be uniform, as LISP demonstrates: from the fact that A is accessible from B it does not follow that B is accessible from A.

Second — and this is a major point, with which we will grapple considerably in our considerations of semantics — structural field elements *are taken to be significant* — it is for this reason that we tend to call them *symbols.* We count as computational, in particular, only those processes consisting of ingredient structures and events to which we, as external observers, attribute semantical import.

The reason that I do not consider a car to be a computer, although I *am* tempted to think of its electronic fuel injection module computationally, arises exactly from this question of the attribution of significance. The main constituents of a car I understand in

terms of mechanics — forces and torques and plasticity and geometry and heat and combustion and so on. These are not *interpreted* notions: the best explanation of a car does not posit an externally attributed semantical intepretation function in order to make sense of the car's internal interactions. With respect to any computer, however, — whether it is an abacus, a calculuator, an electronic fuel injection system, or a full-scale digital computer — the best explanation is exactly in terms of the *interpretation* of the ingredients, even though the machine itself is not allowed access to that interpretation (for fear of violating the doctrine of mechanism). Thus I may know that the ALU in my machine works in such and such a way, but I understand its workings in terms of addition, logical operations, and so forth, *all of which are interpretations* of how it works. In other words the proper use of the term "computational" is as a predicate on explanations, not on artefacts.

The third constraint follows directly on the second: in spite of this semantical attribution, the interior processes of a computational process must interact with these structures and symbols and other processes *in complete ignorance and disregard of any externally attributed semantical weight.* This is the substance of the claim that computation is *formal* symbol manipulation — that computation has to do with the interaction with symbols solely in virtue of their shape or spelling. We within computer science are so used to this formality condition — this requirement that computation proceed syntactically — that we are liable to forget that it is a major claim, and are in danger of thinking that the simpler phrase "symbol manipulation" *means* formal symbol manipulation. But in spite of its familiarity, part of our semantical reconstruction will argue that we have not taken this attribution seriously enough.

A book should be written on all these matters; we mention them here only because they will play an important role in our reconstruction of LISP. There are obvious parallels and connections to be explored, for example, between this external attribution of significance to the ingredients of a computational process, and the question of what would be required for a computational system to be *semantically original* in the sense discussed at the beginning of the previous section. This is not the place for such investigations, although we will make explicit this attribution of significance to LISP structures in our presentation of a full declarative semantics for LISP, as section 1.d and chapter 3 will make clear. The present moral is merely that this attribution is neither something new, nor something specific to LISP's circumstances. The external attribution of significance is a

foundational part of computer science.

1.d. The Rationalisation of Computational Semantics

From even the few introductory sections that have been presented so far, it is clear that semantical vocabulary will permeate our analysis. In discussing the knowledge representation and reflection hypotheses, we talked of symbols that *represented* knowledge about the world, and of structures that *designated* other structures. In the model of computation just presented, we said that the attribution of semantic significance to the ingredients of a process was a distinguishing mark of computer science. Informally, no one could possibly understand LISP without knowing that the atom T stands for truth and NIL for falsity. From the fact that computer science is thought to involve *formal* symbol manipuation we admit not only that the subject matter includes *symbols*, but also that the computations over them occur in explicit ignorance of their semantical weight (you cannot treat a non-semantical object, such as an eggplant or a waterfall, *formally*; simply by using the term *formal* you admit that you attribute significance to it on the side). Even at the very highest levels, when say that a process — human or computational — is reasoning *about* a given subject, or reasoning *about* its own thought processes, we implicate semantics, for the term "semantics" can in viewed, at least in part, as a fancy word for *aboutness*. It is necessary, therefore, to set straight our semantical assumptions and techniques, and to make clear what we mean when we say that we will subject our computational dialects to semantical scrutiny.

1.d.i. Pre-Theoretic Assumptions

In engaging in semantical analysis, our goal is *not* simply to provide a mathematically adequate specification of the behaviour of one or more procedural calculi — one that would enable us, for example, to prove programs correct, given some specification of what they were designed to do. In particular, by "semantics" we do not simply mean a mathematical formulation of the properties of a system, formulated from a meta-theoretic vantage point (unfortunately it seems that the term may be acquiring this rather weak connotation with some writers). Rather, we take semantics to have fundamentally to do with meaning and reference and so forth — whatever they come to — emerging from the paradigmatic human use of language (as we mentioned in section 1.b.i). We are interested in semantics for two reasons: first, because, as we said at the end of the last section, all

computational systems are marked by external semantical attribution, and second, because semantics is the study that will reveal what a computational system is reasoning *about*, and a theory of what a computational process is reasoning about is a pre-requisite to a proper characterisation of reflection.

Given this agenda, we will approach the semantical study of computational systems with a rather precise set of guidelines. Specifically, we will require that our semantical analyses answer to the following two requirements, emerging from the two facts about processes and structural fields laid out at the end of section 1.c:

1. They should manifest the fact that we understand computational structures in virtue of attributing to them semantical import;
2. They should make evident that, in spite of such attribution, computational processes are *formal*, in that they must be defined over structures independent of their semantical weight;

Strikingly, from just these two principles we will be able to defend our requirement of a double semantics, since the attributed semantics mentioned in the first premise includes not only a pre-theoretic understanding of what *happens* to computational symbols, but also a *pre-computational* intuition as to what those symbols *stand for*. We will therefore have to make clear the *declarative* semantics of the elements of (in our case) the LISP structural field, as well as establishing their *procedural* import.

We will explore these results in more detail below, but in its barest outlines, the form of the argument is quite simple. Most of the results are consequences of the following basic tenet (we have relativised the discussion to LISP, for perspicuity, but the same would hold for any other calculus):

> *What LISP structures mean is not a function of how they are treated by the LISP processor; rather, how they are treated is a function of what they mean.*

For example, the expression "(+ 2 3)" in LISP evaluates to 5; the undeniable *reason* is that "(+ 2 3)" is understood as a complex name of the number that is the successor of 4. We arrange things — we defined LISP in the way that we did — so that the numeral 5 is the value *because we know what (+ 2 3) stands for.* To borrow a phrase from Barwise and Perry, our reconstruction is an attempt to *regain our semantic innocence* — an innocence that still permeates traditional formal systems (logic, the λ-calculus, and so forth), but that has been lost in the attempt to characterise the so-called "semantics" of computer

programming languages.

That "(+ 2 3)" designates the number five is self-evident, as are many other examples on which we will begin to erect our denotational account. For example, we have already mentioned the unarguable fact that (at least in certain contexts) T and NIL designate Truth and Falsity. Similarly, it is commonplace use the term "CAR" as a *descriptive function* to designate the first element of a pair, as for example in the English sentence "did you notice that the CAR of that list is the atom LAMBDA". From such practice we have incontrovertible evidence that a term such as (CAR X) *designates* the CAR of the list or pair designated by X. Finally, it is hard to imagine an argument against our assumption that (QUOTE X) designates X (in spite of often-heard claims that QUOTE is a function that *holds off the evaluator*, rather than that it is a naming primitive). In sum, formulating the declarative semantics of a computational formalism is not difficult, once one recognises that it is an important thing to do.

1.d.ii. Semantics in a Computational Setting

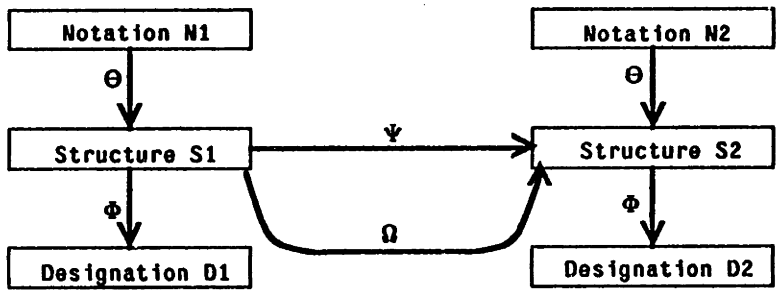
In the most general form that we will use the term *semantics*,[12] a semantical investigation aims to characterise the relationship between a *syntactic* domain and a *semantic* domain — a relationship that is typically studied as a mathematical function mapping elements of the first domain into elements of the second. We will call such a function an ***interpretation*** function (to be sharply distinguished from what in computer science is called an *interpreter*, which we are calling a *processor*). Schematically, as shown in the following diagram, the function Φ is an interpretation function from S to D:

(S1-10)

In a computational setting, this simple situation is made more complex because we are studying a variety of interacting interpretation functions. In particular, the diagram below identifies the relationships between the three main semantical functions that permeate our analysis. Θ is the interpretation function mapping notations into elements of the structural field, Φ is the interpretation function making explicit our attributed semantics to structural field elements, and Ψ is the function formally computed by the language processor. Ω will be explained below; it is intended to indicate a Φ-semantic characterisation of the

relationship between S1 and S2, whereas Ψ indicates the formally computed relationship — a distinction similar, as we will soon argue, to that between the logical relationships of *derivability* (⊢) and *entailment* (⊨).

(S1-11)



For mnemonic convenience, we use the name "Ψ" by analogy with *psychology*, since a study of Ψ is a study of the internal relationships between symbols, all of which are within the machine (Ψ is meant to signify psychology *narrowly construed*, in the sense of Fodor, Putnam, and others[13]). The label "Φ", on the other hand, chosen to suggest *philosophy*, signifies the relationship between a set of symbols and the world.

As an example to illustrate S1-11, suppose we accept the hypothesis that people represent English sentences in an internal mental language we will call mentalese (suppose, in other words, that we accept the hypothesis that our minds are computational processes). If you say to me the phrase "a composer who died in 1750" and I respond with the name "J. S. Bach", then, in terms of the figure, the first phrase, *qua* sentence of English, would be N1; the mentalese representation of it would be S1, and the person who lived in the 17th and 18th century would be the referent D1. Similarly, my reply would be N2, and the mentalese fragment that I presumably accessed in order to formulate that reply would be S2. Finally, D2 would again be the long-dead composer; thus D1 and D2, in this case, would be the same fellow.

N1, N2, S1, S2, D1, and D2, in other words, need not necessarily all be distinct: in a variety of different circumstances two or more of them may be the same entity. We will examine cases, for example, of self-referential designators, where S1 and D1 are the same object. Similarly, if, on hearing the phrase "the pseudonym of Samuel Clemens", I reply "Mark Twain", then D1 and N2 are identical. By far the most common situation, however, will be as in the Bach example, where D1 and D2 are the same entity — a circumstance where we say that the function Ψ is ***designation-preserving***. As we will see in the next

section, the α-reduction and β-reduction of the λ-calculus, and the derivability relationship ($\vdash$) of logic, are both designation-preserving relationships. Similarly, the 2- and 3-LISP processors will be designation-preserving, whereas 1-LISP's and SCHEME's evaluation processors, as we have already indicated, are not.

In the terms of this diagram, the argument we will present in chapter 3 will proceed roughly as follows. First we will review logical systems and the λ-calculus, to show the general properties of the Φs and Ψs employed in those formalisms, for comparison. Next we will shift towards computational systems, beginning with PROLOG, since it has evident connections to both declarative and procedural traditions. Finally we will take up LISP. We will argue that it is not only coherent, but in fact natural, to define a declarative Φ for LISP, as well as a procedural Ψ. We will also sketch some of the mathematical characterisation of these two interpretation functions. It will be clear that though similar in certain ways, they are nonetheless crucially distinct. In particular, we will be able to show that 1-LISP's Ψ (EVAL) obeys the following equation. We will say that any system that satisfies this equation has the ***evaluation property***, and the statement that, for example, the equation holds of 1-LISP the ***evaluation theorem***. (The formulation used here is simplified for perspicuity, ignoring contextual relativisation; S is the set of structural field elements.)

$$\forall S \in S\ [\ \textit{if}\ \Phi(S) \in S\ \textit{then}\ \Psi(S) = \Phi(S)\ \textit{else}\ \Phi(\Psi(S)) = \Phi(S)\ ] \qquad \text{(S1-12)}$$

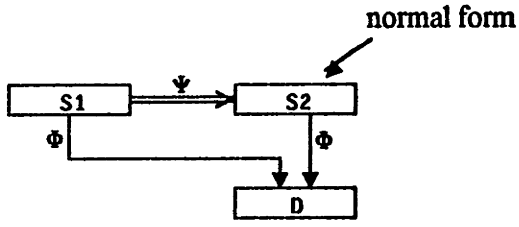
1-LISP's evaluator, in other words, *de-references just those terms whose referents lie within the structural field*, and is designation-preserving otherwise. Where it can, in other words, 1-LISP's Ψ implements Φ; where it is not, Ψ is *Φ-preserving*, although what it does do with its argument in this case has yet to be explained (saying that it preserves Φ is too easy: the identity function preserves designation was well, but EVAL is not the identity function).

The behaviour described by S1-12 is unfortunate, in part because the question of whether $\Phi(S) \in S$ is not in general decidable, and therefore even if one knows of two expressions S_1 and S_2 that S_1 is $\Psi(S_2)$, one still does not necessarily know the relationships between $\Phi(S_1)$ and $\Phi(S_2)$. More seriously, it makes the explicit use of meta-structural facilities extraordinarily awkward, thus defeating attempts to engender reflection. We will argue instead for a dialect described by the following alternative (again in skeletal form):

$$\forall S \in \mathcal{S}\ [[\Phi(S) = \Phi(\Psi(S))] \wedge \text{NORMAL-FORM}(\Psi(S))] \qquad \text{(S1-13)}$$

When we prove it for 2-LISP, we will call this equation the ***normalisation theorem***; any system satisfying we will say has the ***normalisation property***. Diagrammatically, the circumstance it describes is pictured as follows:

(S1-14)



Such a Ψ, in other words, is *always* Φ-preserving. It relies, in addition, on a notion of *normal form*, which we will have to define.

In the λ-calculus, Ψ(S) would *definitionally* be in normal-form, since the concept of that name is *defined* in terms of the non-applicability of any further β-reductions. As we will argue in more detail in chapter 3, this makes the notion less than ideally useful; in designing 2-LISP and 3-LISP, therefore, we will in contrast define normal-formedness in terms of the following three (provably independent) properties:

1. They must be ***context-independent***, in the sense of having the same declarative and procedural import independent of their context of use;
2. They must be ***side-effect free***, implying that their procedural treatment will have no affect on the structural field or state of the processor;
3. They must be ***stable***, by which we mean that they must normalise to themselves in all contexts.

It will then require a proof that all 2-LISP and 3-LISP results (all expressions Ψ(S)) are in normal-form. In addition, from the third property, plus this proof that the range of Ψ includes only normal-form expressions, we will be able to show that Ψ is *idempotent*, as was suggested earlier ($\Psi = \Psi \circ \Psi$, or equivalently, $\forall S\ \Psi(S) = \Psi(\Psi(S))$) — a property of 2-LISP and 3-LISP that will ultimately be shown to have substantial practical benefits.

There is another property of normal-form designators in 2-LISP and 3-LISP, beyond the three requirements just listed, that will follow from our category alignment mandate. In designing those dialects we will insist that the *structural category* of each normal form designator be determinable from *the type of object designated*, independent of the structural

type of the original designator, and independent as well of any of the machinery involved in implementing Ψ (this is in distinction to the received notion of normal form employed in the λ-calculus, as will be examined in a moment). For example, we will be able to demonstrate that any term that designates a number will be taken by Ψ into a numeral, since numerals will be defined as the normal-form designators of numbers. In other words, from just the designation of a term X the *structural category* of Ψ(X) will be predictable, independent of the form of X itself (although the *token identity* of Ψ(X) cannot be predicted on such information alone, since normal-form designators are not necessarily unique or canonical). This category result, however, will have to be proved: we call it the ***semantical type theorem***.

That normal form designators cannot be canonical arises, of course, from computability considerations: one cannot decide in general whether two expressions designate the same function, and therefore if normal-form function designators *were* required to be unique it would follow that expressions that designated functions could not necessarily be normalised. Instead of pursuing that sort of unhelpful approach, we will instead adopt a non-unique notion of normal-form function designator, still satisfying the three requirements specified above: such a designator will by definition be called a *closure*. All well-defined function-designating expressions, on this scheme, will succumb to a standard normalisation procedure.

Some 2-LISP (and 3-LISP) examples will illustrate all of these points. We include the numbers in our semantical domain, and have a syntactic class of *numerals*, which are taken to be normal form number designators. The numerals are canonical (one per number), and as usual they are side-effect free and context independent; thus they satisfy the requirements on normal-formedness. The semantical type theorem says that any term that designates a number will normalise to a numeral: thus if X designates five and Y designates six, and if + designates the addition function, then we know (can prove) that (+ X Y), since it designates eleven, will normalise to the numeral 11. Similarly, there are two boolean constants $T and $F that are normal-form designators of Truth and Falsity, and a canonical set of rigid structure designators called handles that are normal-form designators of all s-expressions (including themselves). And so on: closures are normal-form function designators, as mentioned in the last paragraph; we will also have to specify normal-form designators for sequences and other types of mathematical objects included in the

semantical domain.

We have diverted our discussion away from general semantics, onto the particulars of 2-LISP and 3-LISP, in order to illustrate how the semantical reconstruction we endorse would impinge on a language design. However it is important to recognise that the behaviour mandated by S1-13 is not *new*: this is how all standard semantical treatments of the λ-calculus proceed, and the designation-preserving aspect of it is approximately true of the inference procedures for logical systems as well, as we will see in detail in chapter 3. Neither the λ-calculus reduction protocols, in other words, nor any of the typical inference rules one encounters in mathematical or philosophical logics, *de-reference* the expressions over which they are defined. In fact it is hard to imagine *defending* S1-12. What may have happened, we can speculate, is that because LISP includes its syntactic domain within the semantic domain — because LISP has QUOTE as a primitive operator, in other words — a semantic inelegance was inadvertantly introduced in the design of the language that has never been corrected. Thus our rationalisation of LISP is an attempt to *regain the semantical clarity* of predicate logic and the λ-calculus, in part by connecting the language of our computational calculi with the language in which those prior linguistic systems have been studied.

It is this regained coherence that, we claim, is a necessary prerequisite to a coherent treatment of reflection.

A final comment. The consonance of S1-13 with standard semantical treatments of the λ-calculus, and the comments just made about LISP's inclusion of QUOTE, suggest that one way to view our project is as a semantical analysis of a variant of the λ-calculus with quotation. In the LISP dialects we consider, we will retain sufficient machinery to handle side effects, but it is of course always possible to remove such facilities from a calculus. Similarly, we could remove the numerals and atomic function designators (i.e. the ability to name composite expressions as unities). What would emerge would be a semantics for a deviant λ-calculus with some operator like QUOTE included as a primitive syntactic construct — a semantics for a *meta-structural* extension of the already *higher-order* λ-calculus. We will not pursue this line of attack in this dissertation, but, once the mathematical analysis of 2-LISP is in place, such an analysis should emerge as a straightforward corrollary.

1.d.iii. Recursive and Compositional Formulations

The previous sections have suggested briefly the work that we would like our semantics to do; they do not reveal how this is to be accomplished. In chapter 3, where the reconstruction of semantics is laid out, we of course pursue this latter question in detail, but we can summarise some of its results here. Beginning very simply, standard approaches suffice. For example, we begin with *declarative import* (Φ), and initially posit the designation of each primitive object type (saying for instance that the numerals designate the numbers, and that the primitively recognised closures designate a certain set of functions, and so forth), and then specify recursive rules that show how the designation of each composite expression emerges from the designation of its ingredients. Similarly, in a rather parallel fashion we can specify the *procedural consequence* (Ψ) of each primitive type (saying in particular that the numerals and booleans *are self-evaluating*, that atoms evaluate to their bindings, and so forth), and then once again specify recursive rules showing how the *value* or *result* of a composite expression is formed from the results of processing its constituents.

If we were considering only purely extensional, side-effect free, functional languages, the story might end there. However there are a variety of complications that will demand resolution, of which two may be mentioned here. First, none of the LISP's that we will consider are purely extensional: there are intensional constructs of various sorts (QUOTE, for example, and even LAMBDA, which we will view as a standard intensional procedure, rather than as a syntactic mark). The hyper-intensional QUOTE operator is not in itself difficult to deal with, although we will also consider questions about less-fine grained intensionality of the sort that (a statically scoped) LAMBDA manifests. As in any system, the ability to deal with intensional constructs will cause a reformulation of the entire semantics, with extensional procedures recast in appropriate ways. This is a minor complexity, but no particular difficulty emerges.

The second difficulty has to do with side-effects and contexts. All standard model-theoretic techniques of course allow for the general fact that the semantical import of a term may depend in part of on the context in which it is used (variables are the classic simple example). However the question of side-effects — which are part of the total *procedural consequence* of an expression, impinges on the appropriate context *for declarative*

purposes as well as well as for procedural. For example, in a context in which X is bound to the numeral 3 and Y is bound to the numeral 4, it is straightforward to say that the term (+ X Y) designates the number seven, and returns the numeral 7. However consider the more semantics of the more complex (this is standard LISP):

```
(+ 3 (PROG (SETQ Y 14) Y))                                    (S1-16)
```

It would be hopeless (to say nothing of false) to have the formulation of declarative import ignore procedural consequence, and claim that S1-16 designates seven, even though it patently returns the numeral 17 (although note that we are under no pre-theoretic obligation to make the declarative and procedural stories cohere — in fact we will reject 1-LISP exactly because they do *not* cohere in any way that we can accept). On the other hand, to *include* the procedural effect of the SETQ within the specification of Φ would seem to violate the ground intuition which argued that the designation of this term, and the structure to which it evaluates, are different.

The approach we will ultimately adopt is one in which we define what we call a *general significance function* Σ, which embodies both declarative import (designation), local procedural consequence (what an expression evaluates to, to use LISP jargon), and full procedural consequence (the complete contextual effects of an expression, including side-effects to the environment, modifications to the field, and so forth). Only the total significance of our dialects will be strictly *compositional*; the components of that total significance, such as the designation, will be *recursively specified* in terms of the designation of the consituents, relativised to the total context of use specified by the encompassing function. In this way we will be able to formulate precisely the intuition that S1-16 designates seventeen, as well as returning the corresponding numeral 17.

Lest it seem that by handling these complexities we have lost any incisive power in our approach, we should note that it is not always the case that the processing of a term results in the obvious (i.e., normal-form) designator of its referent. For example, we will prove that the expression

```
(CAR '(A B C))                                                (S1-16)
```

both designates *and* returns the atom A. Just from the contrast between these two examples (S1-15 and S1-16) it is clear that LISP processing and LISP designation do not track each other in any trivially systematic way.

Although our approach will prove successful, we will ultimately abandon the strategy of characterising the full semantics of standard LISP (as exemplified in our 1-LISP dialect), since the confusion about the semantic import of evaluation will in the end make it virtually impossible to say anything coherent about designation. This, after all, is our goal: to *judge* 1-LISP, not merely to *characterise* it. By the time our semantical analysis is concluded, we will not only know *that* LISP is confusing, we will also have seen in detail *why* it is confusing, and we will be adequately prepared to design a dialect that corrects its errors.

1.d.iv. The Role of a Declarative Semantics

One brief final point about this double semantics must be brought out. It should be clear that it is impossible to specify a normalising processor without a pre-computational theory of semantics. If you do not have an account of what structures mean, *independent* of how they are treated by the processor, there is no way to say anything substantial about the semantical import of the function that the processor computes. On the standard approach, for example, it is impossible to say that the processor is correct, or semantically coherent, or semantically incoherent, or anything: it is merely what it is. Given some account of what it does, one can compare this to *other* accounts: thus it is possible for example to prove that a *specification* of it is correct, or that an *implementation* of it is correct, or that it has certain other independently definable properties (such as that it always terminates, or uses certain resources in certain ways). In addition, *given* such an account, one can prove properties of programs written in the language — thus, from a mathematical specification of the processor of ALGOL, plus the listing of an ALGOL program, it might be possible to prove that that program met some specifications (such as that it sorted its input, or whatever). However none of these questions are the question we are trying to answer; namely: *what is the semantical character of the processor itself?*

In our particular case, we will be able to specify the semantical import of the function computed by LISP's EVAL (this is content of the evaluation theorem), but only by first laying out both declarative and procedural theories of LISP. Again, we will be able to design 2-LISP only with reference to this pre-computational theory of declarative semantics. It is a simple point, but it is important to make clear how our semantical reconstruction is a *prerequisite* to the design of 2-LISP and 3-LISP, not a post-facto method of analysing them.

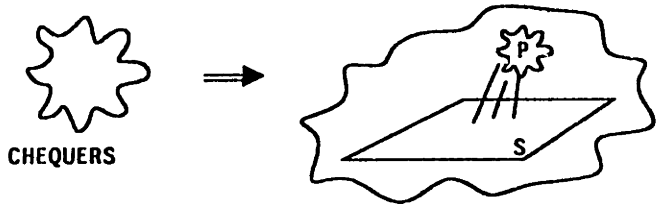
1.e. Procedural Reflection

Now that we have assembled a minimal vocabulary with which to talk about computational processes and matters of semantics, we can sketch the architecture of reflection that we will present in the final chapter of the dissertation. We will start rather abstractly, with the general sense of reflection sketched in section 1.b; we will then make use of both the knowledge representation hypothesis and the reflection hypothesis to define a much more restricted goal. Next, we will employ our characterisations of interpretively reduced computational processes and of computational semantics to narrow this goal even further. As this progressive focussing proceeds, it will become more and more clear what would be be involved in actually constructing an authentically reflective computational language. By the end of this section we will be able to suggest the particular structure that, in chapter 5, we will embody in 3-LISP.

1.e.i. A First Sketch

We begin very simply. At the outset, we characterised reflection in terms of a process shifting between a pattern of reasoning about some world, to reasoning reflectively about its thoughts and actions in that world. We said in the knowledge representation hypothesis that the only current candidate architecture for a process that reasons *at all* (even derivatively) is one constituted in terms of an interior process manipulating representations of the appropriate knowledge of that world. We can see in terms of the process reduction model of computation a little more clearly what this means: for the process we called CHEQUERS to reason about the world of finance, we suggested that it be *interpretively composed* of an ingredient process P manipulating a structural field S consisting of representations of check-books, credit and debit entires, and so forth. Thus we were led to the following picture:

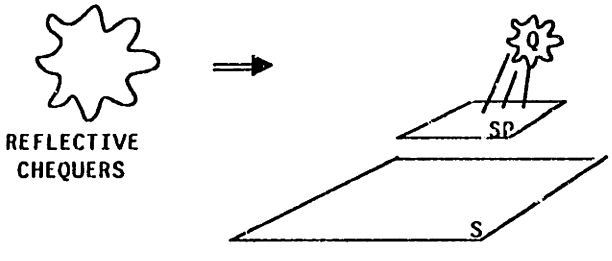
(S1-17)



Next, we said (in the reflection hypothesis) that the only suggestion we have as to how to make CHEQUERS reflective was this: as well as constructing process P to deal with these various financial records, we could also construct process Q to deal with P *and the structural field it manipulates.* Thus Q might specify what to do when P failed or encountered an unexpected situation, based on what parts of P had worked correctly and what state I was in when the failure occured. Alternatively, Q might describe or generate parts of P that hadn't been fully specified. Finally, Q might effect a more complex interpretation process for P, or one particularized to suit specific circumstances. In general, whereas the world of P — the domain that P models, simulates, reasons about — is the world of finance, the world of Q is the world of the process P and the structural field it computes over.

We have spoken as if Q were a *different* process from P, but whether it is really different from P, or whether it is P in a different guise, or P at a different time, is a question we will defer for a while (in part because we have said nothing about the individuation criteria on processes). All that matters for the moment is that there be *some* process that does what we have said that Q must do.

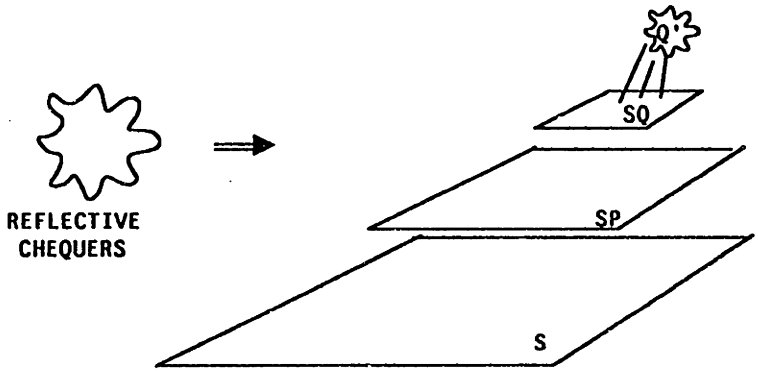
What do we require in order for Q to reason about P? Because Q, *like all the processes we are considering,* is assumed to be interpretively composed, we need what we always need: structural representations of the facts about P. What would such representations be like? First, they must be expressions (statements), formulated with respect to some theory, of the state of process P (we begin to see how the *theory relative* mandate on reflection from section 1.b is making itself evident). Second, in order to actually describe P, they must be *causally connected* to P in some appropriate way (another of our general requirements). Thus we are considering a situation such as that depicted in the following diagram, where the field (or field fragment) SP contains these causally connected structural descriptions:



(S1-18)

This diagram is of course incomplete, in that it does not suggest how SP should relate to P (an answer to this question is our current quest). Note however that reflection must be able to recurse, implying as well something of the following variety:

(S1-19)



Where then might an encodable procedural theory come from? We have two possible sources: in our reconstruction of a semantical analysis we will have presented a full theory of the dialects we will study; this is one candidate for an appropriate theory. Note, however, since we are considering only *procedural* reflection, that although in the general case we would have to encode the full theory of computational significance, in the present circumstance the simpler procedural component will suffice.

The second source of a theoretical account, which is actually quite similar in structure, but even closer to the one we will adopt, is what we will call the *meta-circular processor*, which we will briefly examine.

1.e.ii. Meta-Circular Processors

In any computational formalism in which programs are accessible as first class structural fragments, it is possible to construct what are commonly known as *meta-circular interpreters:* "meta" because they operate on (and therefore terms within them designate) other formal structures, and "circular" because they do not constitute a definition of the processor, for two reasons: they have to be run by that processor in order to yield any sort of behaviour (since they are *programs*, not *processors*, strictly), and the behaviour they would thereby engender can be known only if one knows beforehand what the processor does. Nonetheless, such processors are often pedagogically illuminating, and they will play a critical role in our development of the reflective model. In line with our general strategy

of reserving the word "interpret" for the semantical interpretation function, we will call such processors *meta-circular processors*.

In our presentation of 1-LISP and 2-LISP we will construct meta-circular processors (or *MCP's*, for short); the 2-LISP version is presented here (all the details of what this means will be explained in chapter 4; at the moment we mean only to illustrate the general structure of this code):

```
(DEFINE NORMALISE                                                    (S1-20)
   (LAMBDA EXPR [EXP ENV CONT]
      (COND [(NORMAL EXP) (CONT EXP)]
            [(ATOM EXP) (CONT (BINDING EXP ENV))]
            [(RAIL EXP) (NORMALISE-RAIL EXP ENV CONT)]
            [(PAIR EXP) (REDUCE (CAR EXP) (CDR EXP) ENV CONT)])))

(DEFINE REDUCE                                                       (S1-21)
   (LAMBDA EXPR [PROC ARGS ENV CONT]
      (NORMALISE PROC ENV
         (LAMBDA EXPR [PROC!]
            (SELECTQ (PROCEDURE-TYPE PROC!)
               [IMPR (IF (PRIMITIVE PROC!)
                         (REDUCE-IMPR PROC! ARGS ENV CONT)
                         (EXPAND-CLOSURE PROC! ARGS CONT))]
               [EXPR (NORMALISE ARGS ENV
                        (LAMBDA EXPR [ARGS!]
                           (IF (PRIMITIVE PROC!)
                               (REDUCE-EXPR PROC! ARGS! ENV CONT)
                               (EXPAND-CLOSURE PROC! ARGS! CONT))))]
               [MACRO (EXPAND-CLOSURE PROC! ARGS
                         (LAMBDA EXPR [RESULT]
                            (NORMALISE RESULT ENV CONT)))])))))

(DEFINE EXPAND-CLOSURE                                               (S1-22)
   (LAMBDA EXPR [CLOSURE ARGS CONT]
      (NORMALISE (BODY CLOSURE)
                 (BIND (PATTERN CLOSURE) ARGS (ENV CLOSURE))
                 CONT)))
```

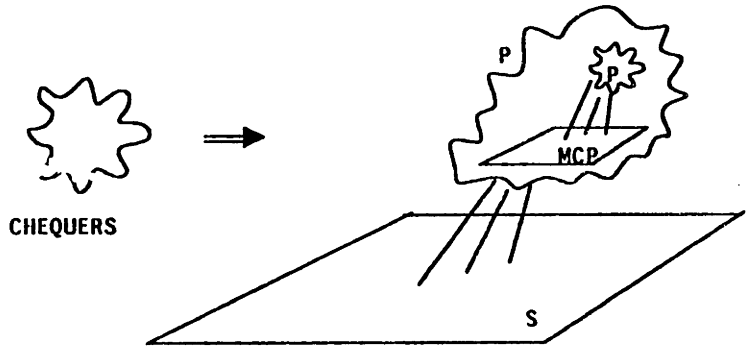
The basic idea is that if this code were processed by the primitive 2-LISP processor, the process that would thereby be engendered would be behaviourally equivalent to that of the primitive processor itself. If, in other words, we were to assume mathematically that processes are functions from structure onto behaviour, and if we called the processor presented as S1-20 through S1-22 above by the name MCP_{2L}, and called the primitive 2-LISP processor P_{2L}, then we would presumably be able to prove the following result, where by "$\simeq$" we mean *behaviourally equivalent*, in some appropriate sense (this is the sort of proof of correctness one finds in for example Gordon:[14]

$$P_{2L}(MCP_{2L}) \simeq P_{2L} \qquad \text{(S1-23)}$$

It should be recognised that the equivalence of which we speak here is a global equivalence; by and large the primitive processor, and the processor resulting from the explicit running of the MCP, cannot be arbitrarily mixed (as a detailed discussion in chapter 5 will make clear). For example, if a variable is bound by the underlying processor P_{2L}, it will not be able to be looked up by the meta-circular code. Similarly, if the meta-circular processor encounters a control structure primitive, such as a THROW or a QUIT, it will not cause the meta-circular processor itself to exit prematurely, or to terminate. The point, rather, is that if an entire computation is mediated by the explicit processing of the MCP, then the results will be the same as if that entire computation had been carried out directly.

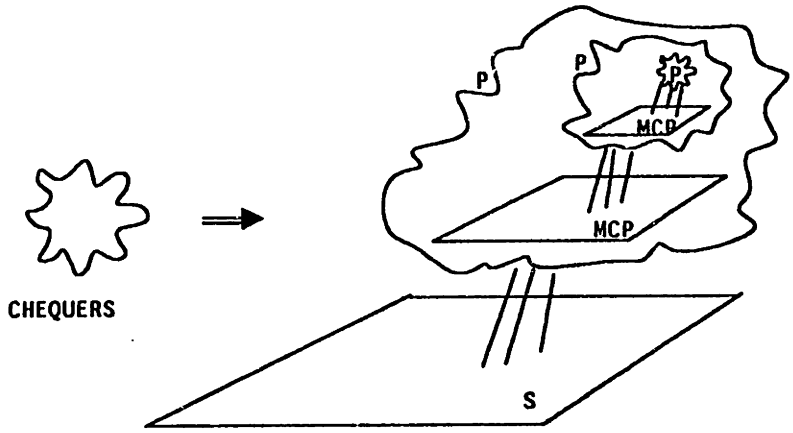
We can merge these results about MCPs with the diagram in S1-17, as follows: if we replaced P in S1-17 with a process that resulted from P processing the meta-circular processor, we would still correctly engender the behaviour of CHEQUERS:

(S1-24)



Furthermore, this replacement could also recurse:

(S1-26)



Admittedly, under the standard interpretation, each such replacement would involve a dramatic increase in inefficiency, but the important point for the time being is that the resulting behaviour would in some sense still be correct.

1.e.iv. Procedural Reflective Models

We can now unify the suggestion made at the end of section 1.e.ii on having Q reflect upwards, with the insights embodied in the MCP's of the previous section, and thereby define what we will call the ***procedural reflective model.*** The fundamental insight arises from the eminent similarity between diagrams S1-18 and S1-19, on the one hand, compared with S1-24 and S1-26, on the other. These diagrams do not represent exactly the same situation, of course, but the approach will be to converge on a unification of the two.

We said earlier that in order to satisfy the requirements on the Q of section 1.e.ii we would need to provide a causally connected structural encoding of a procedural theory of our dialect (we will use LISP) within the accessible structural field. In the immediately preceding section we have seen something that is *appoximately* such an encoding: the meta-circular processor. However (and here we refer back to the six properties of reflection given in section 1.b) in the normal course of events the MCP lacks the appropriate causal access to the state of P: whereas any possible state of Q *could* be procedurally encoded in terms of the meta-circular process (i.e., given any account of the state of P we could retroactively construct appropriate arguments for the various procedures in the meta-circular processor so that if that meta-circular processor were run with those arguments it would mimic P in the given state), in the normal course of events the state of P will *not* be so

encoded.

This similarity, however, does suggest the form of our solution. Suppose first that P were never run directly, but were *always* run in virtue of the explicit mediation of the meta-circular processor — as, for example, in the series of pictures given in S1-24 and S1-25. Then at any point in the course of the computation, if that running of one level of the MCP were *interrupted*, and the arguments being passed around were used by some *other procedures*, they would be given just the information we need: causally connected and correct representations of the state of the process P prior to the point of reflection (of course the MCP would have to be modified slightly in order to support such a protocol of interruption).

The problem with this approach, however, is the following: if we always run P mediated by the meta-circular processor, it would seem that P would be unnecessarily inefficient. Also, this proposal would seem to deal with only one level of reflection; what if the code that was looking at these structural encodings of P's state were themselves to reflect? This query suggests that we have an infinite regress: not only should the MCP be used to run the base level Q programs, but the MCP should be used to run the MCP. In fact *all* of an infinite number of MCP's should be run by yet further MCPs, ad infinitum.

Leaving aside for a moment the obvious vicious regress in this suggestion, this is not a bad approach. The *potentially* infinite set of reflecting processes Q are almost indistinguishable in basic structure from the infinite tower of MCP's that would result. Furthermore the MCP's would contain just the correct structurally encoded descriptions of processor state. We would still need the modification so that some sort of interruption or reflective act could make use of this tower of processes, but it is clear that to a first approximation this solution has the proper character.

Furthermore, it will turn out that we can simply posit, essentially, that the primitive processor *is* engendered by an infinite number of recursive instances of the MCP, each running a version one level below. The implied infinite regress is after all not problematic, since only a finite amount of information is encoded in it (all but a finite number of the bottom levels each MCP is merely running a copy of the MCP). Because we (the language designers) know exactly how the language runs, and know as well what the MCP is like, we can provide this infinite number of levels, to use the current jargon, *only virtually*. As

chapter 5 will explain in detail, such a virtual simulation is in fact perfectly well-defined. It is no longer reasonable to call the processor a *meta-circular* processor, of course, since it becomes inextricably woven into the fundamental architecture of the language (as will be explained in detail in chapter 5). It is for this reason that we will call it the ***reflective processor***, as suggested above. Nonetheless its historical roots in the meta-circular processor should be clear.

In order to ground this suggestion in a little more detail, we will explain just briefly the alteration that allows this architecture to be used. More specifically, we will in 3-LISP support what we will call ***reflective procedures*** — procedures that, when invoked, are run not at the level at which the invocation occured, but one level higher, being given as arguments those expressions that would have been passed around in the reflective processor, had it always been running explicitly. We present the code for the 3-LISP reflective processor here, to be contrasted only very approximately with S1-20 through S1-22 (the important line is underlined for emphasis):

```
(DEFINE NORMALISE                                                    (S1-26)
   (LAMBDA SIMPLE [EXP ENV CONT]
      (COND [(NORMAL EXP) (CONT EXP)]
            [(ATOM EXP) (CONT (BINDING EXP ENV))]
            [(RAIL EXP) (NORMALISE-RAIL EXP ENV CONT)]
            [(PAIR EXP) (REDUCE (CAR EXP) (CDR EXP) ENV CONT)])))

(DEFINE REDUCE                                                       (S1-27)
   (LAMBDA SIMPLE [PROC ARGS ENV CONT]
      (NORMALISE PROC ENV
         (LAMBDA SIMPLE [PROC!]
            (SELECTQ (PROCEDURE-TYPE PROC!)
               [REFLECT ((SIMPLE . ↓(CDR PROC!)) ARGS ENV CONT)]
               [SIMPLE (NORMALISE ARGS ENV (MAKE-C1 PROC! CONT))])))))

(DEFINE MAKE-C1                                                      (S1-28)
   (LAMBDA SIMPLE [PROC! CONT]
      (LAMBDA SIMPLE [ARGS!]
         (COND [(= PROC! ↑REFERENT)
                (NORMALISE ↓(1ST ARGS!) ↓(2ND ARGS!) CONT)]
               [(PRIMITIVE PROC!) (CONT ↑(↓PROC! . ↓ARGS!))]
               [$T (NORMALISE (BODY PROC!)
                              (BIND (PATTERN PROC!) ARGS! (ENV PROC!))
                              CONT)])))))
```

What is important about the underlined line is this: when a redex (application) is encountered whose CAR normalises to a reflective procedure, as opposed to a standard procedure (the standard ones are called SIMPLE in this dialect), the corresponding function

(designated by the abstruse term (SIMPLE . ↓(CDR PROC!)), but no matter) is run *at the level of the reflective processor*, rather than *by* the processor. In other words ths single underlined line in S1-27 on its own unleashes the full infinite reflective hierarchy.

Coping with that hierarchy will occupy part of chapter 5, where we explain all of this in more depth. Just this much of an introduction, however, should convey to the reader at least a glimpse of how reflection is possible.

1.e.iv. Two Views of Reflection

The reader will note a certain tension between two ways in which we have characterised this form of reflection. On the one hand we sometimes speak as if there were a primitive and noticeable *reflective act*, which causes the processor to *shift levels* rather markedly (this is the explanation that best coheres with *some* of the pre-theoretic intuitions about reflective thinking in the sense of contemplation). On the other hand, we have also just spoken of an infinite number of levels of reflective processors, each essentially implementing the one below, so that it is not coherent either to ask at which level Q is running, or to ask how many reflective levels are running: in some sense they are all running at once, in exactly the same sense that both the LISP processor inside your editor, and your editor, are both running when you use that editor. In the editor case it is not, of course, as if LISP and editor were both running *together*, in the sense of *side-by-side* or *independently*, rather, the one, being interior to the other, in fact supplies the anima or agency of the outer one. It is just this sense in which the higher levels in our reflective hierarchy are always running: each of them is in some sense *within* the processor at the level below, so that it can thereby engender it.

We will not take a principled view on which account — a single locus of agency stepping between levels, or an infinite hierarchy of simultaneous processors — is correct: they turn out, rather curiously, to be behaviourally equivalent. For certain purposes one is simpler, for others the other.

To illustrate the "shifting levels" account (which is more complex than the infinite number of levels story), we present the following account of what is involved in constructing a reflective dialect, in part by way of review, and in part in order to suggest to the reader how it is that a reflective dialect could in fact be finitely constructed. In

particular, you have to provide a complete theory of the given calculus expressed within its own language (the reflective processor — this is required on both accounts, obviously). Secondly, you have to arrange it so that, when the process reflects, all of the structures used by the reflective processor (the formal structures designating the theoretical entities posited by the theory) are available for inspection and manipulation, and correctly encode the state that the interpreter was in prior to the reflective jump. Third, you have to make sure that when the process decides to "drop down again", the original base-level interpretation process is resumed in accordance with the facts encoded in those structures. In the minimal case, upon reflection the processor would merely interpret the reflective processor explicitly, then at some further point would drop down and resume running non-reflectively. Such a situation, in fact, is *so* simple that it could not be distinguished (except perhaps in terms of elapsed time) from pure non-reflective interpretation.

The situation, however, would get more complex as soon as the user is given any power. Two provisions in particular are crucial. First, the entire purpose of a reflective dialect is to allow the user to have his or her own programs run along with, or in place of, or between the steps of, the reflective processor. We need in other words to provide an abstract machine with the ability for the programmer to insert code — in convenient ways and at convenient times — at any level of the reflective hierarchy. For example, suppose that we wish to have a λ-expression closed only in the dynamic environment of its use, rather than in the lexical environment of its definition. The reflective model will of course contain code that performs the default lexical closure. The programmer can assume that the reflective code is being explicitly interpreted, and can provide, for the lambda expression in question, an alternate piece of code in which different action is taken. By simply inserting this code into the correct level, (s)he can use variables bound by the reflective model in order to fit gracefully into the overall regime. Appropriate hooks and protocols for such insertion, of course, have to be provided, but they have to be provided only once. Furthermore, the reflective model will contain code showing how this hook is normally treated.

As well as providing for the arbitrary interpretation of special programs, at the reflective level, we need in addition to enable the user to *modify* the explicitly available structures that were provided by the reflective model. Though this ability is easier to design than the former, its correct implementation is considerably trickier. An example will

make this clear. In the LISP reflective model we will exhibit, the interpreter will be shown to deal explicitly with both environment and continuation structures. Upon reflecting, programs can at will access these structures that, at the base level, are purely implicit. Suppose that the user's reflective code actually modifies the environment structure (say to change the binding of a variable in some procedure somewhere up the stack, in the way that a debugging package might support), and also changes the continuation structure (designator of the continuation function) so as to cause some function return to bypass its caller. When this reflective code "returns", so to speak, and drops back down, the interpretation process that must then take effect must be the one mandated by these modified structures, not the one that would have been resumed prior to the reflection. These modifications, in other words, must be noticed. This is the *causal connection* aspect of self-reference that is so crucial to true reflection.

1.e.v. Some General Comments

The details of this architecture emerged from detailed considerations; it is interesting to draw back and see to what extent its global properties match our pre-theoretic intuitions about reflection. First, we can see very simply that it honours all six requirements laid out in section 1.b.iii:

1. It is causally connected and theory-relative;
2. It is theory-relative;
3. It involves an incremental "stepping back" rather than a true (and potentially vicious) instantaneous self-reference;
4. Finer-grained control is provided over the processing of lower level structures;
5. It is only partially detached (3-LISP reflective procedures are still in 3-LISP, they are still animated by the same fundamental agency, since if one level stops processing the reflective model (or some analogue of it), all the processors "below" it cease to exist); and
6. The reflective powers of 3-LISP are primitively provided.

Thus in this sense we can count our architecture a success.

Regarding other intuitions, such as the locus of self, the concern as to whether the potential to reflect requires that one always participate in the world indirectly rather than directly, and so forth, turn out to be about as difficult to answer for 3-LISP as they are to answer in the case of human reflection. In particular, our solution does not answer the

question we posed earlier about the identity of the reflected processor: is it P that reflects, or is it another process Q that reflects on P? The "reflected process" is neither quite the same process, nor quite a different process; it is in some ways as different as an *interior* process, except that since it shares the same structural field it is not as different as an implementing process. No answer is forthcoming until we define much more precisely what the criteria of individuation on processes are, and, perhaps more strikingly, there seems no particular reason to answer the question one way or another. It is tempting (if dangerous) to speculate that the reason for these difficulties in the human case is exactly why they do not have answers in the case of 3-LISP: they are not, in some sense, "real" questions. But it is premature to draw this kind of parallel; our present task is merely to clarify the structure of proposed solution.

1.f. The Use of LISP as an Explanatory Vehicle

There are any number of reasons why it is important to work with a specific programming language, rather than abstractly and in general (for pedagogical accessibility, as a repository for emergent results, as an example to test proposed technical solutions, and so forth). Furthermore, commonsense considerations suggest that a familiar dialect, rather than a totally new formalism, would better suit our purposes. On the other hand, there are no current languages which are categorically and semantically rationalised in the way that our theories of reflection demand; therefore it is not an option to endow any extant system with reflective capabilities, without first subjecting it to substantial modification. It would be possible simply to present some system embodying all the necessary modifications and features, but it would be difficult for the reader to sort out which architectural features were due to what concern. In this dissertation, therefore, we have adopted the strategy of presenting a reflective calculus in two steps: first, by modifying an existing language to conform to our semantical mandates, and second, by extending the resulting rationalised language with reflective capabilities.

Once we have settled on this overall plan, the question arises as to what language should be used as a basis for this two-stage development. Since our concern is with *procedural* rather than with general reflection, the class of languages that are relevant includes essentially all programming languages, but excludes exemplars of the declarative tradition: logic, the λ-calculus, specification and representation languages, and so forth (it is important to recognise that the suggestion of constructing a reflective variant of the λ-calculus represents a category error). Furthermore, we need a programming language — a procedural calculus — with at least the following properties:

1. The language should be simple; reflection by itself is complicated enough that, especially for the first time, we should introduce it into a formalism of minimal internal complexity.
2. It must be possible to access program structures as first-class elements of the structural field.
3. Meta-structural primitives (the ability to *mention* structural field elements, such as data structures and variables, as well as to *use* them) must be provided.

4. The underlying architecture should facilitate the embedding, within the calculus, of the procedural components of its own meta-theory.

The second property could be added to a language: we could devise a variant on ALGOL, for example, in which ALGOL programs were an extended data type, but LISP already possesses this feature. In addition, since we will use an extended λ-calculus as our meta-language, it is natural to use a procedural calculus that is functionally oriented. Finally, although full-scale modern LISPs are as complex as any other languages, both LISP 1.5 and SCHEME have the requisite simplicity.

LISP has other recomendations as well: because of its support of accesssible program structures, it provides considerable evidence of exactly the sort of inchoate reflective behaviour that we will want to reconstruct. The explicit use of EVAL and APPLY, for example, will provide considerable fodder for subsequent discussion, both in terms of what they do well and how they are confused. In chapter 2, for example, we will describe a half dozen types of situation in which a standard LISP programmer would be tempted to use these meta-structural primitives, only two of which in the deepest sense have to do with the explicit manipulation of expressions; the other four, we will argue, ought to be treated directly in the object language. Finally, of course, LISP is the *lingua franca* of the AI community; this fact alone makes it an eminent candidate.

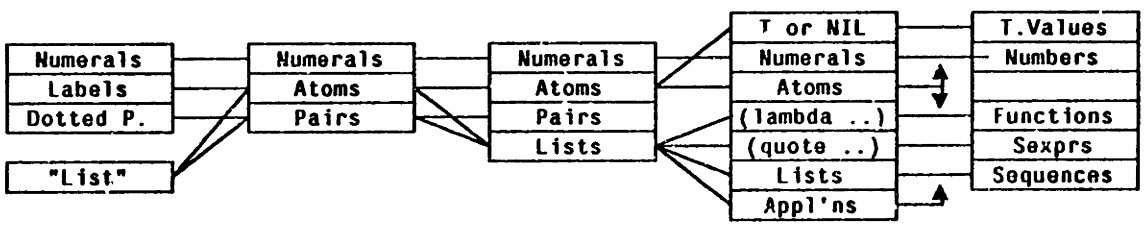
1.f.i. 1-LISP as a Distillation of Current Practice

The decision to use LISP as a base doesn't solve all of our problems, since the name "LISP" still refers to rather a wide range of languages. It has seemed simplest to define a simple kernel, not unlike LISP 1.5, as a basis for further development, in part to have a fixed and well-defined target to set up and criticise, and in part so that we could collect into one dialect the features that will be most important for our subsequent analysis. We will take LISP 1.5 as our primary source, although some facilities we will ultimately want to examine as examples of reflective behaviour — such as CATCH and THROW and QUIT — will be added to the repertoire of behaviours manifested in McCarthy's original design. Similarly, we will include macros as a primitive procedure type, as well as intensional and extensional procedures of the standard variety ("call-by-value" and "call-by-name", in standard computer science parlance, although we will avoid these terms, since we will reject the notion of "value" entirely).

It will not be entirely simple to present 1-LISP, given our theoretical biases, since so much of what we will ultimately reject about it comes so quickly to the surface in explaining it. However it is important for us to present this formalism without modifying it, because of the role it is to play in the structure of our overall argument. We ask of the dialect not that it be clean or coherent, but rather that it serve as a vehicle in which to examine a body of practice suitable for subsequent reconstruction. To the extent that we make empirical claims about our semantic reconstructions, we will point to 1-LISP practice (our model for all standard LISP practice) as evidence. *Therefore, for theoretical reasons, it is crucial that we leave that practice intact and free of our own biases.* Thus, we will uncritically adopt, in 1-LISP, the received notions of evaluation, lists, free and global variables, and so forth, although we will of course be at considerable pains to document all of these features rather carefully.

As an example of the style of analysis we will engage in, we present here a diagram of the category structure of 1-LISP that we will formulate in preparation for the category alignment mandate dominating 2-LISP:

(S1-29)



The intent of the diagram is to show that in 1-LISP (as in any computational calculus) there are a variety of ways in which structures or s-expressions may be categorised; the point we are attempting to demonstrate is the (unnecessary) complexity of interaction between these various categorical decompositions.

In particular, we may just briefly consider each of these various 1-LISP categorisations. The first (*notational*) is in terms of the lexical categories that are accepted by the reader (including strings that are parsed into notations for numerals, lexical atoms, and "list" and "dotted-pair" notations for pairs). Another (*structural*) is in terms of the primitive types of s-expression (numerals, atoms, and pairs); this is the categorisation that is typically revealed by the primitive structure typing predicates (we will call this procedure

TYPE in 1-LISP, but it is traditionally encoded in an amalgam of ATOM and NUMBERP). A third traditional categorisation (*derived structural*) includes not only the primitive s-expression types but also the derived notion of a *list* — a category built up from some pairs (those whose CDRs are, recursively, lists) and the atom NIL. A fourth taxonomy (*procedural consequence*) is embodied by the primitive processor: thus 1-LISP's EVAL sorts structures into various categories, each handled differently. This is the "dispatch" that one typically finds at the top of the meta-circular definition of EVAL and APPLY. There are usually six discriminated categories: i) the self-evaluating atoms T and NIL, ii) the numerals, iii) the other atoms, used as variables or global function designators, depending on context, iv) lists whose first element is the atom LAMBDA, which are used to encode applicable functions, v) lists whose first element is the atom QUOTE, and vi) other lists, which in evaluable positions represent function application. Finally, the fifth taxonomy (*declarative import*) has to do with declarative semantics — what categories of structure *signify* different sorts of semantical entities. Once again a different category structure emerges: applications and variables can signify semantical entities of arbitrary type *except that they cannot designate functions* (since 1-LISP is first-order); the atoms T and NIL signify Truth and Falsity; general lists (including the atom NIL) can signify enumerations (sequences); the numerals signify numbers; and so on and so forth.

Any reflective program in 1-LISP would have to know about all of these various different categorisations, and about the relationships between them (as presumably all human LISP programmers do). We need not dwell on the obvious fact that confusion is a likely outcome of this categorical disarray.

One other example of 1-LISP behaviour will be illustrative. We mentioned above that 1-LISP requires the explicit use of APPLY in a variety of circumstances; these include the following:

1. When an argument expression designates a function *name*, rather than a function (as for example in `(APPLY (CAR '(+ - *)) '(2 3))`).
2. When the arguments to a multi-argument procedure are designated by a single term, rather than individually (thus if `X` evaluates to the list `(3 4)`, one must use `(APPLY '+ X)` rather than `(+ X)` or `(+ . X)`).
3. When the function is designated by a variable rather than by a global constant (thus one must use `(LET ((FUN '+)) (APPLY FUN '(1 2)))` rather than `(LET ((FUN '+)) (FUN 1 2))`).

4. When the arguments to a function are "already evaluated", since APPLY, although itself extensional (is an EXPR), does not re-evaluate the arguments even if the procedure being applied is an EXPR (thus one uses (APPLY '+ (LIST X Y)), rather than (EVAL (CONS '+ (LIST X Y)))).

As we will see below, in 2-LISP (and 3-LISP) only the first of these will require explicitly mentioning the processor function by name, because it deals inherently with the designation of expressions, rather than with the designation of their referents. The other three will be adequately treated in the object language

1.f.ii. The Design of 2-LISP

Though it meets our criterion of simplicity, 1-LISP will provide more than ample material for further development, as the previous two examples will suggest. Once we have introduced it, we will, as mentioned earlier, subject it to a semantical analysis that will lead us into an examination of computational semantics in general, as described in the previous section. The search for semantical rationalisation, and the exposition of the 2-LISP that results, will occupy a substantial part of the dissertation, even though the resulting calculus will still fail to meet the requirements of a procedurally reflective dialect. We discussed what semantic rationalisation comes to in the previous section; here will sketch how its mandates are embodied in the design of 2-LISP.

The most striking difference between 1-LISP and 2-LISP is that the latter rejects evaluation in favour of independent notions of *simplification* and *reference*. Thus, 2-LISP's processor is not called EVAL, but NORMALISE, where by *normalisation* we refer to a particular form of expression simplification that takes each structure into what we call a *normal-form* designator of that expression's referent (normalisation is thus designation preserving). The details will emerge in chapter 4, but a sense of the resulting architecture can be given here.

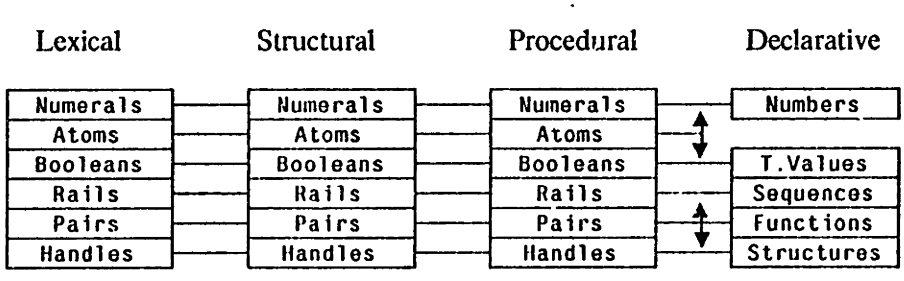
Simple object level computations in 2-LISP (those that do not involve meta-structural terms designating other elements of the LISP field) are treated in a manner that looks very similar to 1-LISP. The expression (+ 2 3), for example, normalises to 5, and the expression (= 2 3) to $F (the primitive 2-LISP boolean constant designating falsity). On the other hand an obvious superficial difference is that *meta-structural* terms are not automatically de-referenced. Thus the quoted term 'X, which in 1-LISP would evaluate to X, in 2-LISP normalises to itself. Similarly, whereas (CAR '(A . B)) would evaluate in 1-LISP to A, in 2-

LISP it would normalise to 'A; (CONS 'A 'B) would evaluate in 1-LISP to (A . B); in 2-LISP the corresponding expression would return '(A . B).

From these trivial examples, an ill-advised way to think of the 2-LISP processor emerges: as if it were just like the 1-LISP processor except that it puts a quote back on before returning the result. In fact, however, the differences are much more substantial, in terms of both structure, procedural protocols, and semantics. For one thing 2-LISP is statically scoped (like SCHEME) and higher-order (function-designating expressions may be passed as regular arguments). Structurally 2-LISP is also rather different from 1-LISP: there is no derived notion of *list*, but rather a primitive data structure called a *rail* that serves the function of designating a sequence of entities (pairs are still used to encode function applications). So called "quoted expressions" are primitive, not applications in terms of a QUOTE procedure, and they are canonical (one per structure designated). The notation 'X, in particular, is not an abbreviation for (QUOTE X), but rather the primitive notation for a *handle* that is the unique normal-form designator of the atom X. There are other notational differences as well: rails are written with square brackets (thus the expression "[1 2 3]" notates a rail of three numerals that designates a sequence of three numbers), and expressions of the form "(F A_1 A_2 ... A_k)" expand not into "(F . (A_1 . (A_2 . (... . (A_k . NIL)...))))" but into "(F . [A_1 A_2 ... A_k])".

The category structure of 2-LISP is summarised in the following diagram:

(S1-30)



Closures, which have historically been treated as rather curiously somewhere between functions and expressions, emerge in 2-LISP as standard expressions; in fact we *define* the term "closure" to refer to a *normal-form function designator.* Closures are pairs, and all normal-form pairs are closures, illustrating once again the category alignment that permeates the design.

All 2-LISP normal-form designators are not only *stable* (self-normalising), but are also *side-effect free* and *context-independent.* A variety of facts emerge from this result. First, the primitive processor (NORMALISE) can be proved to be *idempotent*, in terms of both result and total effect:

```
∀S [(NORMALISE S) = (NORMALISE (NORMALISE S))]                    (S1-31)
```

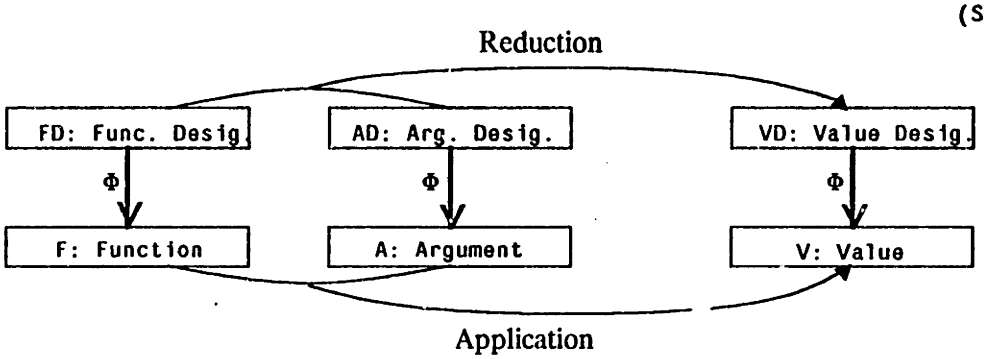
Consequently, as in the λ-calculus, the result of normalising a constituent (in an extensional context) in a composite expression can be substituted back into the original expression, in place of the un-normalised expression, yielding a partially simplified expression that will have the same designation and same normal-form as the original. In addition, in code-generating code such as macros and debuggers and so forth there is no need to worry about whether an expression has *already* been processed, since second and subsequent processings will never cause any harm (nor, as it happens, will they take substantial time).

Much of the complexity in defining 2-LISP will emerge only when we consider forms that designate other semantically significant forms. The intricacies of just such "level-crossing" expressions form the stock-in-trade of a reflective system designer, and only by setting such issues straight *before* we consider reflection proper will we face the latter task adequately prepared. Primitive procedures called NAME and REFERENT (abbreviated as "↑" and "↓") are provided to mediate betweeen sign and significant (they must be primitive because otherwise the processor remains semantically flat); thus (NAME 3) normalises to '3, and (REFERENT ''A) to 'A.

The issue of the explicit use of "APPLY", which we mentioned briefly in discussing 1-LISP above, is instructive to examine in 2-LISP, since it manifests both the structural and the semantic differences between 2-LISP and its precursor dialect. In 1-LISP, the two functions EVAL and APPLY mesh in a well-known mutually-recursive fashion. Evaluation is uncritically thought to be defined over *expressions*, but it is much less clear what application is defined over. On one view, "apply" is a functional that maps functions and (sequences of) arguments onto the value of the function at that argument position — thus making it a second (or higher) order function. On another, "apply" takes two *expressions* as arguments, and has as a value a third expression that designates the value of the function designated by the first argument at the argument position designated by the second. In 2-LISP we will call the first of these *application* and the second *reduction* (the latter in part

because the word suggests an operation over expressions, and in part by analogy with the β-reduction of Church[15]). Current LISP systems are less than lucid regarding this distinction (in MACLISP, for example, the *function* argument is an expression, whereas the *arguments* argument is not, and the value is not). The position we will adopt is depicted in the following diagram (which we will explain more fully in chapter 3):

(S1-32)



The procedure REDUCE, together with NORMALISE will of course play a major role in our characterisation of 2-LISP, and in our construction of the reflective 3-LISP. However it will turn out that there is no reason to define a designator of the APPLY function, since any term of the form

```
(APPLY FUN ARGS)                                    (S1-33)
```

would be entirely equivalent in effect to

```
(FUN . ARGS)                                        (S1-34)
```

REDUCE, in contrast, since it is a meta-structural function, is neither trivial to define (as APPLY is) nor recursively empty.

A summary of the most salient differences between 2-LISP and 1-LISP is provided in the following list:

1. 2-LISP is lexically scoped, in the sense that variables free in the body of a LAMBDA form take on the bindings in force in their statically enclosing context, rather than from the dynamically enclosing context at the time of function application.
2. Functions are first-class semantical objects, and may be designated by standard variables and arguments. As a consequence, the function position in an application (the CAR of a pair) is normalised just as other positions are.

3. Evaluation is rejected in favour of independent notions of *simplifcation* and *reference*. The primitive processor is a particular kind of *simplifier*, rather than being an *evaluator*. In particular, it *normalises* expressions, returning for each input expression a normal-form co-designator.
4. A complete theory of declarative semantics is postulated for all s-expressions, prior to and independent of the specification of how they are treated by the processor function (this is a pre-requisite to any claim that the processor is designation-preserving).
5. Closures — normal-form function designators — are valid and inspectable s-expressions.
6. Though not all normal-form expressions are canonical (functions, in particular, may have arbitrarily many distinct normal-form designators), nevertheless they are all *stable* (self-normalising), *side-effect free*, and *context independent*.
7. The primitive processor (NORMALISE) is *semantically flat*; in order to shift level of designation one of the explicit semantical primitives NAME or REFERENT must be applied.
8. 2-LISP is *category-aligned* (as indicated in S1-30 above): there are two distinct structural types, *pairs* and *rails*, that respectively encode function applications and sequence enumerations. There is in addition a special two-element structural class of boolean constants. There is no distinguished atom NIL.
9. Variable binding is *co-designative*, rather that *designative*, in the sense that a variable *normalises* to what it is *bound to*, and therefore designates the referent of the expression to which it is bound. Though we will speak of the *binding* of a variable, and of the *referent* of a variable, we will not speak of a variable's *value*, since that term is ambiguous between these two.
10. Identity considerations on normal-form designators are as follows: the normal-form designators of truth-values, numbers, and s-expressions (i.e., the booleans, numerals, and handles) are unique; normal-form designators of sequences (i.e., the rails) and of functions (the pairs) are not. No atoms are normal-form designators; therefore the question does not arise in their case.
11. The use of LAMBDA is purely an issue of abstraction and naming, and is completely divorced from procedural *type* (extensional, intensional, macro, and so forth).

As soon as we have settled on the definition of 2-LISP, however, we will begin to criticise it. In particular, we will provide an analysis of how 2-LISP fails to be reflective, in spite of its semantical cleanliness. A number of problems in particular emerge as troublesome. First, it will turn out that the clean *semantical* separation between meta-levels is not yet matched with a clean *procedural* separation. For example, too strong a separation between environments, with the result that intensional procedures become extremely

difficult to use, shows that, in one respect, 2-LISP's inchoate reflective facilities suffer from insufficient causal connection. On the other hand, awkward interactions between the control stacks of inter-level programs will show how, in other respects, there is *too much* connection. In addition, we will demonstrate a meta-circular implementation of 2-LISP in 2-LISP, and we will provide 2-LISP with explicit names for its basic interpreter functions (NORMALISE and REDUCE). However these two facilities will remain *utterly unconnected* — an instance of a general problem we will have discussed in chapter 3 on reflection in general.

1.f.iii. The Procedurally Reflective 3-LISP

From this last analysis will emerge the design of 3-LISP, a procedurally reflective LISP and the last of the dialects we will consider. 3-LISP, presented in chapter 5, differs from 2-LISP in a variety of ways. First, the fundamental *reflective act* is identified and accorded the centrality it deserves in the underlying definition. Each reflective level is granted its own environment and continuation structure, with the environments and continuations of the levels below it accessible as first-class objects (meriting a Quinean stamp of ontological approval, since they can be the values of bound variables). These environments and continuations, as mentioned in the discussion earlier, are theory relative: the (procedural) theory in question is the 3-LISP reflective model, a causally connected variant on the meta-circular interpreter of 2-LISP, discussed in section 1.e. Surprisingly, the integration of reflective power into the meta-circular (now reflective) model is itself extremely simple (although to *implement* the resulting machine is not trivial).

Once all these moves have been taken it will be possible to merge the explicit reflective version of SIMPLIFY and REDUCE, and the similarly named primitive functions. In other words the 3-LISP reflective model unifies what in 2-LISP were separate: primitive names for the underlying processor, and explicit meta-circular programs demonstrating the procedural structure of that processor.

It was a consequence of defining 2-LISP in terms of SIMPLIFY that the 2-LISP interpreter "stays semantically stable": the semantical level of an input expression is always the same as that of the expression to which it simplifies. An even stronger claim holds for function application: except in the case of the functions NAME and REFERENT, the semantical level of the result is also the same as that of all of the arguments. This is all evidence of the attempt to drive a wedge between *simplification* and *de-referencing* that we mentioned

earlier. 3-LISP inherits this semantical characterisation (it even remains true, surprisingly, in the case of reflective functions). A fixed-level interpreter like this — and of course this is one of the reasons we made 2-LISP this way — enables us to make an important move: we can approximately identify *declarative meta levels* with *procedural reflective levels.* This does not quite have the status of a *claim,* because it is virtually mandated by the knowledge representation hypothesis (furthermore, the correspondence is in fact asymetric: declarative levels can be crossed within a given reflective level, but reflective shifts always involve shifts of designation). But it is instructive to realise that we have been able to identify the reflective act (that makes available the structures encoding the interpretive state and so forth) with the shift from objects to their names. Thus what was *used* prior to reflection is *mentioned* upon reflecting; what was *tacit* prior to reflection is *used* upon reflection. When this behaviour is combined with the ability for reflection to recurse, we are able to lift structures that are normally tacit into explicit view in one simple reflective step; we can then obtain access to designators of those structures in another.

Both the 3-LISP reflective model, and a MACLISP implementation of it, will be provided by way of definition. In addition, some hints will be presented of the style of semantical equations that a traditional denotational-semantics account of 3-LISP would need to satisfy, although a full semantical treatment of such a calculus has yet to be worked out. In a more pragmatic vein, however, and in part to show how 3-LISP satisfies many of the *desiderata* that motivated the original definition of the concept of reflection, we will present a number of examples of programs defined in it: a variety of standard functions that make use of explicit evaluation, access to the implementation (debuggers, "single-steppers", and so forth), and non-standard evaluation protocols. The suggestion will be made that the ease with which these powers can be embedded in "pure" programs recommends 3-LISP as a plausible dialect in its own right. Nor is this simply a matter of using 3-LISP as a theoretical vehicle to model these various constructs, or of showing that such models fit naturally and simply into the 3-LISP dialect (as a simple continuation-passing style can for example be shown to be adapted in SCHEME). The claim is stronger: that they can be naturally embedded in a manner that allows them to be congenially mixed (without pre-compilation) with the simpler, more standard practice. Although the user *need not* use an explicit continuation-passing style, nonetheless, at any point in the course of the computation, the continuation is explicitly available (upon reflection) for those programs

that wish to deal with it directly. Similar remarks hold for other aspects of the control structure and environment.

One final comment on the architecture of 3-LISP will relate it to the "two views on reflection" that were mentioned at the end of section 1.e. Interpretation mediated by the 3-LISP reflective model is guaranteed to yield indistinguishable behaviour (at least from a non-reflective point of view — there are subtleties here) from basic, non-reflected interpretation. This fact allows us to posit that 3-LISP runs in virtue of an infinite number of levels of reflective model all running at once, by an (infinitely fleet) overseeing processor. The resulting infinite abstract machine is well defined, for it is of course behaviourally indistinguishable from the perfectly finite 3-LISP we will already have laid out (and implemented). For some purposes this is the simplest way to describe 3-LISP. Since the user can write programs to be interpreted at any of these reflective levels, and cannot tell that all infinitude of levels are not being run (the implementation surreptitiously constructs them and places them in view each time the user's program steps back to view them), such a characterisation is sometimes more illuminating than talk of the processor "switching back and forth from one level to another". It is the goals of modelling psychologically intuitive reflection — based on a vague desire to locate the *self* of the machine at some level or other — that will lead us usually to use the language of explicit shifts (this also more closely mimics the implementation we will have built), although if 3-LISP were to be treated as a purely formal object, the infinite characterisation is probably to be preferred.

1.f.iv. Reconstruction Rather Than Design

2-LISP and 3-LISP can claim to be dialects of LISP only on a generous interpretation. The two dialects are unarguably *more different from the original* LISP 1.6 *than are any other dialects that have been proposed*, including for example SCHEME, MDL, NIL, SEUS, MACLISP, INTERLISP, and COMMON LISP.[16]

In spite of this difference, however, it is important to our enterprise to call these languages LISP. We do not simply propose them as new variants in a grand tradition, perhaps better suited for a certain class of problems than those that have gone before. Rather — and this is one of the reasons that the dissertation is as long as it is — we claim that the architecture of these new dialects, in spite of its difference from that of standard

LISPS, *is a more accurate reconstruction of the underlying coherence that organises our communal understanding of what* LISP *is.* We are making a claim, in other words — a claim that should ultimately be judged as right or wrong. Whether 2-LISP or 3-LISP is *better* than previous LISPS is of course a matter of some interest on its own, but it is not the thesis that this dissertation has set out to argue.

1.g. Remarks

1.g.i. Comparision with Other Work

Although we know of no previous attempts to construct either a semantically rationalised or a reflective computational calculus, the research presented here is of course dependent on, and related to, a large body of prior work. There are in particular four general areas of study with which our project is best compared:

1. Investigations into the meta-cognitive and intensional aspects of problem solving (this includes much of current research in artificial intelligence);
2. The design of logical and procedural languages (including virtually all of programming language research, as well as the study of logics and other declarative calculi);
3. General studies of semantics (including both natural language and logical theories of semantics, and semantical studies of programming languages); and
4. Studies of self-reference, of the sort that have characterised much of meta-mathematics and theory of computability throughout this century particularly since Russell, and including the formal study of the paradoxes, the incompleteness results of Gödel, and so forth.

We will make detailed comments about our connections with such work throughout the discussion (for example in chapter 5 we will compare our notion of self-reference with the traditional notion used in logic and mathematics), but some general comments should be made here.

Consider first the meta-cognitive aspects of problem-solving, of which the dependency-directed deduction protocols presented by Stallman and Sussman, Doyle, McAllester, and others are an illustrative example.[17] This work depends on explicit encodings, in some form of meta-language, of information about object-level structures, used to guide a deduction process. Similarly, the meta-level rules of Davis in his TEIRESIUS system,[18] and the use of meta-levels rules as an aid in planning,[19] can be viewed as examples of inchoate reflective problem solvers. Some of these expressions are primarily procedural in intent,[20] although declarative statements (for example about dependencies) are perhaps more common, with respect to which particular procedural protocols are defined.

The relationship of our project to this type of work is more accurately described as one of support, rather than of direct contribution. We do not present (or even hint at) problem solving strategies involving reflective manipulation, although the fact that others are working in this area is a motivation for our research. Rather, we attempt to provide a rigorous account of the particular issues that have to do simply with providing such reflective *abilities*, independent of what such facilities are then used for. An analogy might be drawn to the development of the λ-calculus, recursive equations, and LISP, in relationship to the use of these formalisms in mathematics, symbolic computation, and so forth: the former projects provide a language and architecture, to be used reliably and perhaps without much conscious thought, as the basis for a wide variety of applications. The present dissertation will be successful not so much if it forces everyone working in meta-cognitive areas to think about the architecture of reflective formalisms, but rather if it allows them to forget that the technical details of reflection were ever considered problematic. Church's α-reduction was a successful manoeuvre precisely because it means that one can treat the λ-calculus in the natural way; we hope that our treatment of reflective procedures will enable those who use 3-LISP or any subsequent reflective dialect to treat "backing-off" in the natural way.

The "reflective problem-solver" reported by Doyle[21] deserves a special comment: again, we provide an underlying architecture which might facilitate his project, without actually contributing solutions to any of his particular problems about how reflection should be effectively used, or when its deployment is appropriate. Doyle's envisaged machine is a full-scale problem solver; it is also (at least so he argues) presumed to be large, to embody complex theories of the world, and so forth. In contrast, our 3-LISP is not a problem solver at all (it is a language very much in need of programming); it embodies only a small procedural theory of itself, and it is really quite small. As well as these differences in goals there are differences in content (we for example endorse a set of reflective levels, rather than any kind of true instantaneous self-referential reflexive reasoning); it is difficult, however, to determine with very much detail what his proposal comes to, since his report is more suggestive than final.

Given that 3-LISP is not a problem solver of the sort Doyle proposes, it is natural to ask whether it would be a suitable language for Doyle to use to *implement* his system. There are two different kinds of answer to this question, depending on how he takes his

project. If he is proposing a design of a complete computational architecture (i.e., a process reduced in terms of an ingredient processor and a structural field), and wishes to *implement* it in some convenient underlying language, then 3-LISP's reflective powers will not in themselves immediately engender corresponding reflective powers in the virtual machine that he implements. Reflection, as we are at considerable pains to demonstrate, is first and foremost a semantical phenomenon, and *semantical properties* — designation and normalisation protocols and reflection and the rest — *do not cross implementation boundaries* (this is one of the great powers of implementation). 3-LISP would be useful in such a project to the extent that it is generally a useful and powerful language, but it is important to recognise that its reflective powers cannot be used directly to provide refle reflective capabilities in other architectures implemented on top of it.

Of course Doyle would have an alternative strategy open to him, by which he could use 3-LISP's reflective powers more directly. If, rather than defending a generic reflective architecture, he more simply intended to show how a particular kind of reflective reasoning was useful, he could perhaps construct such behaviour in 3-LISP, and thus use the reflective capabilities of that dialect rather directly. There are, however, consequences of this approach: he would have to accept 3-LISP structures and semantics, including the fact that it is purely a procedural formalism. It would not be possible, in other words, to encode a full descriptive language on top of 3-LISP, and then use 3-LISP's reflective powers to reflect in a general sense with these descriptive structures. If one aims to construct a general or purely descriptive formalism, one must make that architecture reflective on its own.

None of these conclusions stand as criticisms of 3-LISP; they are entailments of fundamental facts of computation and semantics, not limitations of our particular theory or dialect (i.e., they would be equally true of any other proposed architecture). Furthermore, it is not at this level that our contribution is primarily aimed. What *would* presumably be useful to Doyle (or to anyone else in a parallel circumstance) is the detailed structure of a reflective system that we explicate here — an architecture and a concomitant set of theoretical terms *to help him analyse and structure whatever architecture he adopts.* Thus we might expect him to make use of the Ψ/Φ distinction, the relationship between semantical levels and reflective levels, the encoding of the reflective model within the calculus, the strategy of using a virtually infinite processor in a finite manner, the uniformity of a normalising processor, the elegance of a category-aligned language, and so forth. It is in

this sense that the theory and understanding that 3-LISP embodies would (we hope) contribute to this variety of research, rather than the particular formalism we have demonstrated by way of illustration.

The second type of research with which our project has strong ties is the general tradition of providing formalisms to be used as languages and vehicles for a variety of other projects — from the formal statement of theories, the construction of computational processes, the analysis of human language, and so forth. We include here such a large tradition (including logic and the λ-calculus and virtually all of programming language research) that it might seem difficult to say anything specific, but a variety of comments can be made. First, we of course owe a tremendous debt to the LISP tradition in general,[22] and also to the recent work of Steele and Sussman.[23] Particularly important is their SCHEME dialect — in many ways the most direct precursor of 2-LISP (in an early version of the dissertation I called SCHEME "1.7-LISP", since it takes what I see as half the step from LISP 1.5 to our semantically rationalised 2-LISP). Second, our explicit attempt to unify the declarative and procedural aspects of this tradition has already been mentioned — a project that is (as far as we know) without precedent. The PROLOG calculus,[24] as we mentioned in the introduction, must be discounted as a candidate, since it provides two calculi together, rather than presenting a given calculus under a unified theory. Finally, as documented throughout the text, inchoate reflective behaviour can be found in virtually all corners of computational practice; the SMALLTALK language,[25] to mention just one example, includes a meta-level debugging system that allows for the inspection and incremental modification of code in the midst of a computation.

The third and fourth classes of previous work listed above have to do with general semantics and with self reference. The first of these is considered explicitly in chapter 3, where we compare our approach to this subject with model theories in logic, semantics of the λ-calculus, and the tradition of programming language semantics; no additional comment is required here. Similarly, the relationship between our notions of reflection and traditional concepts of self-reference are taken up in more detail in chapter 5; here we merely comment that our concerns are, perhaps surprisingly, constrained almost entirely to computational formalisms. Unless a formal system embodies a locus of active agency — an internal processor of some sort — the entire question of causal relationship between an encoding of self-referential theory and what we consider a genuine reflective model cannot

even be asked. We often informally think, for example, of a natural deduction "process" or some other kind of deductive apparatus making inferences over first-order sentences — this heuristic makes sense of the formal notion of derivability. Strictly speaking, however, in the purely declarative tradition *derivability* is a simple formal relationship that holds between certain sentence types; no activity is involved. There are no notions of *next* or of *when* a certain deduction is made. *If* one were to specify an active deductive process over such first-order sentences, then it is imaginable that one could include sentences (relative to some axiomatisation of that deductive process) in such a way that the operations of the deductive process were appropriately controlled by those sentences (this is the suggestion we explored briefly in section 1.b.ii). The resulting machine, however — not merely in its reflective incarnation, but even prior to that, by including an active agency — cannot fairly be considered simply a *logic*, but rather a full computational formalism of some sort.

Of course we believe that a reflective version of a descriptive system like this could be build (in fact we intend to construct just such a machine). Our position with respect to such an image rests on two observations: a) it would be an inherently *computational* artefact, in virtue of the addition of independent agency, and b) 3-LISP, although reflective, is not yet such a formalism, since it is purely procedural.

We conclude with one final comparison. The formalism closest in spirit to 3-LISP is Richard Weyhrauch's FOL system,[26] although our project differs in several important technical ways from his. First, FOL, like Doyle's system, is a problem solver: it embodies a theorem-prover, although it is possible (through the use of FOL's meta-levels) to give it guidance about the deduction process. Nevertheless FOL is not a *programming language.* Furthermore, FOL adopts — in fact explicitly endorses — the distinction between declarative and procedural languages (first order logic and LISP, in particular), using the procedural calculus as a *simulation structure* rather than as a descriptive or designational language. Weyhrauch claims that the power that emerges from combining (although maintaining as distinct) these *L-S pairs* ("langauge-simulation-structure" pairs) at each level in his meta hierarchy as one of his primary contributions; it is our claim that the greatest power will arise from dismantling the difference between procedural and declarative calculi. There are other differences as well: the interpretation function that maps terms onto objects in the world outside the computational system (Φ) is crucial to us; it would appear in Weyhrauch's systems as if that particular semantical relationship is abandoned in favour of internal

relationships between one formal system and another. A more crucial distinction is hard to imagine, although there is some evidence[27] that this apparent difference may have to do with our respective uses of terminology, rather than with deep ontological or epistemological beliefs.

In sum, FOL and 3-LISP are technically quite distinct, and the theoretical analyses on which they are based are almost unrelated. Nevertheless at a more abstract level they are clearly based on similar and perhaps parallel, if not identical, intuitions. Furthermore, it is our explicit position that 3-LISP represents merely a first step in the development of a fully reflective calculus based on a fully integrated theory of computation and representation; how such a system would differ from FOL remains to be seen. It seems likely that the resulting unified calculus, rather than the dual-calculus nature, would be the most obvious technical distinction, although the actual structure of the descriptive language, semantical meta-theories, and so forth, may also differ both in substance and in detail.

There is however one remaining difference which is worth exploring in part because it reveals a deep but possibly distinctive character to our treatment of LISP. It is clear from Weyhrauch's system that he considers the procedural formalism to represent a kind of *model* of the world — in the sense of an (abstract) artefact whose structure or behaviour mimics that of some other world of interest. Under this approach the computational behaviour can be taken *in lieu of* or *in place of* the real behaviour in the world being studied. Consider for example the numeral addition that is the best approximation a computer can make to actually adding numbers (whatever that might be). When we type "(+ 1 2)" to a LISP processor and it returns "3" we are liable to take those numerals not so much as *designators* of the respective numbers, but instead as *models*. There is no doubt that the input expression "(+ 1 2)" is a linguistic artefact; on the view we will adopt in this dissertation there is no doubt that the resultant numeral "3" is also a linguistic artefact, but we want to admit here a not unnatural tendency to think of it as standing in place of the actual number, in a different sense from standard designation. It is this sense of *simulation* rather than *description* that underlies Weyhrauch's use of LISP.

It is our belief that this is a limited view, and we go to considerable trouble to maintain an approach in which *all* computational structures are *semantical* in something like a linguistic sense, rather than serving as *models*. There are many issues, having to do with

such issues as truth, completeness, and so forth, that a simulation stance cannot deal with; at worst it leads to a view of computational models in danger of being either radically solipsistic or even nihilist. It is exactly the *connection* between a computational system and the world that motivates our entire approach; a connection that can be ignored only at considerable peril. We in no way rule out computations that in different respects mimic the behaviour of the world they are about: it is clear that certain forms of human anlysis involve just this kind of thinking ("stepping through" the transitions of some mechanism, for examlpe). Our point is merely that such simulation is a kind of thinking about the world; it is not the world being thought about.

1.g.ii. The Mathematical Meta-Language

Throughout the dissertation we will employ an informal meta-language, built up from a rather eclectic combination of devices from quantificational logic, the lambda calculus, and lattice theory, extended with some straightforward conventions (such as expressions of the form "*if P then A else B*" as an abbreviation for "$[P \supset A] \wedge [\neg P \supset B]$"). Notationally we will use set-theoretic devices (union, membership, etc.), but these should be understood as defined over *domains* in the Scott-theoretic sense, rather than over unstructured sets. The notations should by and large be self-explanatory: a few standard conventions worth noting are these:

1. By "$[A \rightarrow B]$" we refer to the domain of continuous functions from A to B;
2. By "$F : [A \rightarrow B]$" we mean that F is a function whose domain is A and whose range is B;
3. By "$\langle S_1, S_2, \ldots, S_k \rangle$" we designate the mathematical sequence consisting of the designata of "S_1", "S_2", ... , and "S_k";
4. By "S^i" we refer to the i'th element of S, assuming that S is a sequence (thus $\langle A,B,C \rangle^2$ is B);
5. By "$[S \times R]$" we designate the (potentially infinite) set of all tuples whose first member is an element of S and whose second member is an element of R;
6. By "A^*" we refer to the power domain of A: $[A \cup [A \times A] \cup [A \times A \times A] \cup \ldots]$.
7. Parentheses and brackets are used interchangeably to indicate scope and function applications in the standard way.
8. We employ standard currying to deal with functions of several arguments. Thus, by "$\lambda A_1, A_2, \ldots, A_k \ . \ E$" and by "$\lambda \langle A_1, A_2, \ldots, A_k \rangle \ . \ E$" we mean "$\lambda A_1 . [\lambda A_2 . [\ldots \ . \ [\lambda A_k \ . \ E] \ldots]]$". Similarly, by "$F(B_1, B_2, \ldots, B_k)$" we mean

"$((\ldots((F(B_1))B_2)\ldots)B_k)$"

If we were attempting to be more precise, we should use domains rather than sets, in order that function continuity be maintained, and so forth. It is not our intent here to make the mathematics rigourous, but it would presumably be straightforward, given the accounts we will set down, to take this extra step towards formal adequacy.

1.g.iii. Examples and Implementations

There are a considerable number of examples throughout the dissertation, which can be approximately divided into two groups: formal statements about LISP and about semantics expressed in the meta-language, and illustrative programs and structures expressed in LISP itself (most of the latter are in one of the three LISP dialects, though there are a few in standard dialects as well). The meta-linguistic characterisations, as the preceding discussion will suggest, have not been checked by formal means for consistency or accuracy; the proofs and derivations were generated by the author using paper and pencil. The programming examples, on the other hand, were all tested on computer implementations of 1-LISP, 2-LISP, and 3-LISP developed in the MACLISP and LISP MACHINE LISP dialects of LISP at M.I.T. (a listing of the third of these is given in the appendix). Thus, although the examples in the text were typed in by the author as text — i.e. the lines of characters in this document are not actual photocopies of computer interaction — nevertheless each was verified by these implementations (furthermore, the implementation presented in the appendix is an actual computer listing). Any residual errors (it is hard to imagine every one has been eliminated) must have arisen either from typing errors or from mistakes in the implementation itself.

Chapter 2. 1-LISP: A Basis Dialect

We will base the technical analysis of subsequent chapters on a "standard" LISP, with which to contrast the reconstructed and reflective dialects we will subsequently design. There are options open regarding such a definition; as has often been remarked, there is some ambiguity as to exactly what the term "LISP" denotes.[1] Though we will initially be unconcerned with issues of programming environments and input/output, and will focus on the basic primitives, we will ultimately want to look at user interaction, since much of how we understand LISP is most clearly revealed there. The most plausible extant candidates are McCarthy's LISP 1.5 and Steele and Sussman's SCHEME. Although LISP 1.5 has history and explicitly formulated semantics on its side,[2] the lexical scoping and "full-funarg"[3] properties of SCHEME recommend it both in terms of theoretical cleanliness and in faithfulness to the λ-calculus. On the other hand SCHEME's partial avoidance of such features as an explicitly available EVAL or APPLY weaken it for our purposes, since such "level-crossing" capabilities are close to our primary subject matter. In addition SCHEME, like LISP, is not a fixed target; various versions have been reported.[4]

There is however a more serious difficulty with SCHEME, relating to our concern with semantics and reflection. As mentioned in the introduction, LISP 1.5 (and therefore all LISPs in current use, since they are all based on it) are essentially *first-order* languages, employing meta-structural machinery to handle what is at heart higher order functionality. In LISP 1.5, for example, expressions that we take to designate functions (like "CONS" and "(LAMBDA ...)") cannot be used in regular argument position, and those functions that would most naturally seem to be defined as higher order functions, like MAP and APPLY, are in fact defined over *expressions*, not over *functions* as such; thus for example in LISP 1.5 we would use

```
(MAPCAR '(LAMBDA (X) (+ X 1)) '(2 3 4))                    (S2-1)
```

rather than

```
(MAPCAR (LAMBDA (X) (+ X 1)) '(2 3 4))                     (S2-2)
```

as a way of producing '(3 4 5), since the first argument to MAPCAR must evaluate to an expression (and designate an expression, although we have no way of saying that yet).

SCHEME, by according functional arguments first class status (S2-2 is a valid SCHEME expression), is, like the λ-calculus, an untyped higher order formalism; unlike the λ-calculus, however, it contains primitive operators (QUOTE, in particular) that make the structural field (the syntactic domain) part of the standard model. LISP 1.5, in other words, is *meta-structural* but *first-order*, whereas the λ-calculus, in symmetric contrast, is *not* meta-structural, but is *higher* order. SCHEME takes a different stand in this space: it is both meta-structural and higher order; this is one of the reasons that it is important, in that it represents a first step towards including both of these functionalities, while maintaining them as distinct. In fact it is plausibly because SCHEME embraces a higher-order base language that it originally omitted the explicit functions EVAL and APPLY, since it is those two functions that enable the LISP 1.5 programmer to mimic higher-order functionality by manipulating expressions in their place (current implementations of SCHEME support EVAL and APPLY, but as "magic forms" like LAMBDA, rather than as first-class procedures, in spite of their being extensional). LISP 1.5, the λ-calculus, and SCHEME, in other words, occupy three points in the four-way classification of programming languages generated by these two binary distinctions; traditional programming languages, of course, are found in the fourth class, since they are typically neither meta-structural nor higher-order. 2- and 3-LISP, like SCHEME, will be meta-structural and higher order. These categorisations are summarised in the following diagram.

(S2-3)

	Meta-Structural	Not Meta-Structural
First Order	LISP 1.5	Standard Programming Languages (ALGOL etc.)
Higher Order	SCHEME, 2-LISP, 3-LISP	The Lambda Calculus

In spite of a certain cleanliness, we will argue that the most natural separation between higher-order functionality and meta-structural powers is not maintained in SCHEME's evaluation process — that this crucial distinction, in other words, is only partially embraced in that dialect. In particular, the separation of *function application* from *expression de-referencing* that arises naturally once one adopts the distinction is not reflected in SCHEME: as we will make clear in chapter 3, SCHEME still *de-references meta-structural expressions upon evaluation* (the λ-calculus has no meta-structural expressions, so the issue does not arise in its case). Since automatic de-referencing is a practice we will argue against, it would be

confusing to base our analysis on a SCHEME-like dialect located half-way between the first-order (meta-structural) position taken by LISP 1.5, and the position that on our view represents that natural semantical position once higher-order functions are admitted. It will, in other words, be easier to show that the SCHEME position is an intermediate one, if that is not where we ourselves begin.

There is another aspect of SCHEME against which we will argue: although it successfully deals with higher-order functionality in the base language — without, that is to say, requiring meta-structural powers — it still requires the use of meta-structural machinery to deal with certain types of objectification and compositionality. For example, in order to apply a function to a sequence of arguments when that sequence is designated by a single expression, rather than by a sequence of expressions, one must resort to the explicit use of APPLY and EVAL — in this respect SCHEME is like traditional LISPs. For example, whereas in LISP 1.5 one would use:

```
(LET ((X '(3 4)))                          ; This is LISP 1.5          (S2-4)
    (APPLY '+ X))
```

in SCHEME, because of its higher-order orientation, you would not have to quote the function desigator, but you would still have to use APPLY:

```
(LET ((X '(3 4)))                          ; This is SCHEME            (S2-5)
    (APPLY + X))
```

We will be able to show how this property results from the lack of category correspondence shared by all these dialects, and will ultimately (in 2-LISP) show how all standard objectifications can be adequately treated without requiring meta-structural designation.

There is yet another advantage of starting with a first-order language. There is a natural connection between the free vaiable scoping protocols of a dialect and its functional "order". Thus we find dynamic variable scoping protocols used in first-order languages that admit the meta-structural treatment of functions, in contrast with, a parallel connection between lexical scoping and the adoption of a higher-order object language. For example, consider the following LISP 1.5 (first-order) definition of a procedure of two arguments — a number and a list — designed to return a list constructed from the second argument, but with each element incremented by the first argument:

```
(DEFINE INCREASE                                                    (S2-6)
   (LAMBDA (NUM LIST)
      (MAPCAR '(LAMBDA (EL) (+ EL NUM)) LIST)))
```

Since MAPCAR requires an *expression* rather than a *function* as its argument, the only way in which this natural use of the bound variable NUM could work is for the dialect to be dynamically scoped. If it were statically (lexically) scoped, the expression passed to MAPCAR would be separated completely from the context in which NUM was bound, and the computation would fail.

In contrast, a higher order dialect such as SCHEME would support the following definition:

```
(DEFINE INCREASE                                                    (S2-7)
   (LAMBDA (NUM LIST)
      (MAPCAR (LAMBDA (EL) (+ EL NUM)) LIST)))
```

In this case, if the dialect were *dynamically* scoped, the binding of NUM would be found so long as MAPCAR did not itself use that variable name, and as long as the function designator (LAMBDA (EL) (+ EL NUM)) was only passed downwards, and so forth.[5] In a statically scoped dialect, however, presumably correct (intended) function is designed in all cases.

It is by no means accidental, in other words, that SCHEME and the λ-calculus are lexically scoped and higher order, whereas all other LISPs are dynamically scoped and first order. There is no theoretical difficulty in defining, say, a lexically-scoped first-order language, but such a calculus would be extremely awkward to use. These issues relate as well to the question of whether the "function position" in an application ("F" in "(F A B C)") is *evaluated*: lexically scoped higher-order languages typically evaluate that position just as they do argument positons; first order languages naturally do not. In addition, the dynamic/lexical distinction relates to the question of what a calculus takes the *intension* of a function to be: dynamic scoping is closely associated with taking it to be an expression (againt consonant with a generally meta-structural stance), whereas lexical scoping associates with taking it to be something more abstract (consonant with a higher-order approach). (Functional intensions are discussed more fully in chapter 4.)

For all of these reasons we will base our progression of LISPs on a simple dynamically-scoped, first-order LISP dialect, called 1-LISP. 1-LISP supports what in MACLISP are called FEXPRS and MACROS, as well as standard EXPRS. We assume, as usual, that the dialect is defined over numbers and truth-values as well as s-expressions (i.e. that

numerals and the boolean constants T and NIL are elements of the 1-LISP structural field). We will adopt the standard LISP practice of representing "applications" (what we will want to define an application to be will be taken up shortly) as *lists*, the first element of which will be taken as signifying (in an as-yet unspecified way) a *function*, and the remaining elements as signifying arguments to that function. This syntactic form will be used in addition for what are called *special forms*[6] such as lambda expressions, quotations, etc., as well as for general enumerations.

In a fuller version of this dissertation it would be appropriate to define 1-LISP completely, introducing function applications, recursion, meta-structural facilities, scoping protocols, and so forth. We will not take up this task here, however, deferring the reader to the literature for most of these preparations. We will in particular assume the discussions of LISP in McCarthy, Allen, Winston, and Weizman, and also the investigations of Sussman and Steele.[7] We will depend particularly on the discussions of meta-circular interpreters and tail-recursion given by Steele and Sussman.[8] What we will do, however, is to characterise the 1-LISP structural field, in order to introduce the way that we will talk about fields in general, and because it will be easiest to describe the 2-LISP and 3-LISP fields with respect to this basis one. This task is taken up below.

As well as using 1-LISP as a base, we will from time-to-time refer to SCHEME — a dialect that supports higher-order functionality, and a concomitant partial separation of meta-structural machinery — in part because the continuation-passing versions of the SCHEME meta-circular interpreter cannot be straightforwardly encoded in a first-order dialect. In order to have a specific and structurally comparable dialect we will use the name "1.7-LISP" for our dialect of SCHEME — structurally identical to 1-LISP, but statically scoped and supporting functional arguments in the SCHEME manner. Thus our trio of dialects is in fact a quartet, with 1.7-LISP/SCHEME sitting slightly to the side, between 1-LISP and 2-LISP. The overall mandate under which all of this is pursued, of course, is one of freeing up the meta-structural capabilities of the calculus for use in reflection, unimpeded by intruding consequences of higher-order functionality and simple objectification. We will show, in other words, that higher-order functionality is not *inherently* a subject requiring meta-structural treatment: it is not in any foundational way an issue of the manipulation of *structures* or *expressions* (as the existence of sound models for the untyped λ-calculus of course has shown). The fact that SCHEME only partially separates the two notions, in other

words, will be shown to be an unnecessary aspect of its design. Reflection, on the other hand, is inherently concerned with expressions and their interpretation, and thus will necessarily involve the use of meta-structural machinery.

We will also depend on a variety of computational concepts and practices that will emerge in subsequent examples. Included will be notions of THROW and CATCH (and other non-local control jumps), the use of continuations, meta-circular interpreters, tail-recursion, programming environment constructs that enable a user to manipulate the stack and environment, and so forth. Most of these are part of the accepted lore in the LISP community; discussions can again be found in the reports of Sussman and Steele.

One final remark. In characterising 1-LISP we must distinguish two kinds of understandings, one a non-computational but powerful conception formulated in terms of *function application*; the other a computational and complete but less convivial account in terms of formal *expression manipulation*, in terms of a depth-first recursive tree walk. It is to LISP's credit that these two kinds of understanding can by and large be allied, but to *confuse* them can lead to misconceptions later in the anlaysis. We will look at these two kinds of understanding in turn.

First, the basic intuition underlying *how we understand the 1-LISP processor* is that it applies functions to arguments, returning their values — this is why LISP is the paradigmatic example of what are called *applicative* languages. For example, the fact that (CAR '(A B)) evaluates to A is typically explained in terms of CAR being a function from pairs to their CARs. Similarly, the expression (+ 2 3) returns 5, because we understand it as representing the application of the addition function to the numbers two and three. Both CAR and + are primitive functions; as well as being provided with this primitive set the programmer is provided with a variety of naming conventions and compositional construction techniques, enabling him to build up what seem to be complex function definitions from simpler ones. For example, the expression

```
(DEFINE INCREMENT (LAMBDA (X) (+ X 1)))                    (S2-8)
```

defines a new function called INCREMENT in terms of the primitive addition function. After this definition has taken effect, the expression (INCREMENT 16) can be viewed as representing the application of this new function to the number 16. In other words, the syntactic methods of defining composite procedures facilitate the user thinking that he or

she is able to describe complex *functions* that, like the primitive ones, can be *applied* to arguments. That this is how we understand LISP procedures is reflected as well in the naturalness of the view reflected in traditional semantics on which s-expressions like CAR and (LAMBDA (X) (+ X 1)) are taken to designate functions.

Like all semantical attribution, however, this taking of expressions to represent the application of functions to arguments is something we external observers do; the LISP processor itself doesn't have any access — nor does it *need* any access — to that significance. Rather, it is defined to perform certain operations in a systematic manner depending on the form of the expression under "interpretation". *It is the expression, not the mathematical function signified,* that drives the interpretation process. In simple cases we can substitute one understanding for another, although, when we get to details, subtleties, or complexities, we often turn to our understanding of *how the interpreter works,* since in in complex cases our basic attributed intuition may fail. The reason is that the underlying intuition of function application, although it permeates our language and practice, is nonetheless *not a computational intuition* — a fact whose importance cannot be overestimated. Function application is not a concept built up out of notions of formal "symbol" manipulation, but rather of *designation* of functional terms and application and so forth: all Platonic and mathematical abstractions. Typically, it is only when it fails (as with side effects, or when dealing with temporal considerations and so forth), or when we need to examine a particular implementation, that we make recourse to a truly computational account.

In sum, function application is not *what the LISP processor actually does*; rather, it is *what we semantically take the LISP processor to do.*

What the 1-LISP processor actually does is of course *formal,* roughly summarisable as follows: a single-locus active agent — a *serial processor* — performs a depth-first recursive tree-walk down "expressions", using non-primitive names that it encounters as standing in place of procedure definitions or values, in various context-dependent ways, ultimately executing the primitive "instructions" or "procedures" whose primitively recognised names are found at the leaves of the resulting tree. The processor merely embodies a controlled set of state-changing operations guided by this recursive-descent control pattern. For example, when the name of a "user-defined function" is encountered

(like the INCREMENT of S2-8), the processor does not figure out what function is signified; rather, it merely looks up the lambda expression associated with that name, and uses that expression ((+ X 1), in our example) to continue its tree walk (subject to certain environment modifications — modifications to its own internal state — which we will presently examine).

As we introduce and explain each of our LISP dialects, we will discuss both the *attributed* kind of understanding and the *formal* way in which the 1-LISP processor works. This double viewpoint, however, should not be confused with the more substantive claim, to be examined chapter 3, that there are *two natural kinds of attributed* understanding. The present claim that there are two different ways to explain 1-LISP, in other words, is not yet the phenomenon mentioned in chapter 1 requiring a double semantics. Rather, our current task is merely to make manifest the primary fact that we understand LISP programs semantically, much in the way in which we understand logical deductions systems semantically, in terms of entailment ($\models$), as well as formally, in terms of derivability ($\vdash$). The arguments for double semantics, and a clarification of the relationship between the formal LISP processor and these semantical treatments, depend on the prior acceptance of the fact that computational systems are quintessentially semantical.

Two additional distinctions, of a very different kind from that between formal and attributed understanding, will organise our presentation of each of the LISPs. The first is the informal separation between *programs* and *data structures* — informal, as mentioned above, because we are not yet able to avail ourselves of the theoretical machinery to make the distinction precise. The second is a three way distinction among the following three kinds of facilities: *primitive* facilities provided by the basic calculus, methods of *composition* enabling the user to construct complex structures and behaviours out of simpler ones, and methods of *abstraction* than enable these composite constructions to be used and refered to as cohesive wholes (mechanisms that make them, in Maturana's phrase,[9] *composite unities*). For example, as well as demonstrating a dozen simply named procedures provided primitively in 2-LISP, we will show how λ-abstraction and recursion can be used to generate more complex procedures (like the (LAMBDA (X) (+ X 1)) of our example), and will show how a variety of naming conventions can be used to allow these complex procedures to be invoked merely by using an atomic name (such as INCREMENT), just as in the case of the primitive ones. Our focus will be on programs, rather than on data structures, but a

parallel development for data structures is possible: we can demonstrate the primitive data structures, show how arrangements of these primitive structures can be welded together into complex composite structures, and show how naming conventions can be used so that these data abstractions can be treated as functional units, again in the same way that primitive data types are utilised as if they were indivisible wholes.

With these preparatory remarks, there remains only the task of characterising the 1-LISP field.

In the sense sketched in section 1.c, to specify a computational calculus is to specify a process functionally analysed in terms of a structural field and the surface behaviour of an interpretive process defined over that field. To review, a structural field is a set of abstract objects, formally defined, standing in some specified set of relationships with one another, over which a locality metric is defined, and with respect to which a set of mutability constraints are specified.

A 1-LISP system consists of a structural field of ***s-expressions*** and a (behaviourally defined) ***1-LISP processor*** (our terminology for what is always called the 1-LISP *interpreter*). S-expressions are of three disjoint kinds: ***atoms*** (atomic elements typically used as names or identifiers), ***numerals*** (also atomic, signifying numbers), and ***pairs***. There are three first-order relationships defined on this field: the ***CAR-relationship*** and the ***CDR-relationship*** (each of which holds between a pair and an arbitrary s-expression), and the ***property-list*** relationship (which holds between an atom and an instance of the derived category of *list*, which we will define below). All three of these relationships are *total functions:* each and every pair has exactly one CAR and one CDR, and each and every atom has one property list. There is a temptation to view pairs as *composite* objects, but that is strictly false, since the identity of the pair is not itself a function of the identity of what would be called its constituents (distinguishable pairs can have the same CAR and the same CDR, and you can change both CAR and CDR without changing the pair).

Two of the three first-order relationships (the CAR and CDR) are *mutable*, in the sense that the relationship between a pair and its CAR can be changed, as can the relationship between a pair and its CDR. The third (the property-list relationship), however, is fixed: one cannot associate a *different* list with an atom. These two mutable relationships are the only mutable aspects of the field — there is no other way in which the field can be changed.

Thus the set of structural field *objects* (the atoms, numerals, and pairs) is constant, and there is no way in which elements can be added or removed (we will deal with CONS in terms of accessibility, not actual creation). The field as a whole, which consists of these objects and of the three relationships (with appropriate constraints on mutability and locality), is subject to change over the course of the computation, in virtue of the interaction of the 1-LISP processor.

The third requirement on specifying a field, after identifying the objects and relationships, and the mutability properties, is to identify the salient *locality* constraints. Locality is always defined over relationships (not objects), of which in 1-LISP we have identified three binary first-order types. 1-LISP has no individual-specific relationships at all, and therefore no individual-specific locality constraints either, which greatly simplifies the analysis. In addition, each of the category-specific locality metrics is assymetric: from a pair both its CAR and CDR are locally accessible, and from an atom its property list is locally accessible, but no one of these relationships is local in the opposite direction. We will be restricted, in defining the surface of the interpretation process, to specify as atomic operations only those that obey these locality considerations.

We cannot (and need not) present a lexical grammar for this field structure, because to do so would associate a notation with the structural field elements, and imply some structure for pairs to indicate their parts, all of which would be misleading.

The foregoing describes only what we will call the ***category*** structure of the 1-LISP field; the ***individual*** structure is as follows: there are twenty-one distinguished (and of course distinct) atoms, called CAR, CDR, CONS, COND, EQ, NUMBERP, QUOTE, ATOM, LAMBDA, READ, PRINT, SET, DEFINE, EVAL, APPLY, +, -, *, /, T, and NIL. These names are for the present simply names we will use to identify them in the text — i.e., names in our theoretical meta-language, which at the present happens to be English; if we were presenting a complete characterisation of 1-LISP we would define them as part of the token structure of 1-LISP *notation.* Even when we introduce labels for them in the notation, however, we will not make those labels (often known as *p-names* for "print names") themselves elements of the structural field, since strings are not (in the present account) a primitively supported data type.

By the *category/individual* distinction we refer implicitly on the one hand to sets of entities taken as a whole, and on the other hand to their individual elements. However we use the term *"category"* to refer not to a set of entities, but rather to *a concept in our theory of* 1-LISP, of which the set is the extension. Thus all the 1-LISP atoms taken together are the extension of the category *atom*; each particular 1-LISP atom is an *individual* atom of this category. The category *atom* is a theoretical abstraction, part not of the 1-LISP structural field but of the theory of 1-LISP we are adopting to describe that field.

This distinction between the concepts of *category* and *individual* is different from that between the notions of *type* and *token*, as those terms are used in theories of language. There is no immediate notion of *type* and *token* in the 1-LISP field, although these notions will impinge on the discussion of notation below (and we will shortly define a derived notion of *type* over pairs, as an extension). In other words, we do not have a notion of an atom *type*, of which there are many tokens, the way that we often speak of a word type (such as the type *orrerry*), and of instances of that type (such as the one in the previous parenthetical fragment of this sentence). If we speak of some atom A, in other words, we refer simply to a single atom: there is no sense to be made of such terminology as *an occurrence of that atom.*

Similarly, all the 1-LISP pairs — all of the elements of the extension of the category *pair* — are distinct individuals, over which it is meaningless to speak of an *occurrence* or *token.* However in the case of pairs (as opposed to atoms and numerals) there is a temptation to define a different notion of *type* or *category*, because of the natural tendency, mentioned above, to think of pairs as approximately composite objects, constituted of two "ingredients": their CAR and their CDR (a temptation re-inforced by the fact that procedural consequence is most naturally defined over such types). On the face of it, this naive intuition could lead to a whole range of *degrees* of type-identity, since two pairs could have the *same* elements, or could have elements that, recursively, were of the same type. We could define a hierarchy of "type-ness" in which distinct pairs whose elements were type-identical of some degree would in turn be type-identical of greater degree. However even this suggestion would need further complication: even if a pair's CAR and CDR were type-identical, they would in general be identical to a different degree, and thus a simple numerical ordering would be insufficiently structured. One would have to define the *degree* of type-identity of two pairs to be the ordered pair of type-identity of their CARs and CDRs.

We will not pursue this fine-grained measure of type-equivalence. However there is a coarser variety of type-identity on pairs that is useful in characterising the 1-LISP procedural component: a notion that is approximately embodied in the standard definition of a LISP identity predicate called EQUAL (in contrast with EQ — the primitive identity predicate over objects in the field). The intuition on which it is based is to say of two pairs that they are type identical just in case the *non-pair terminals in the tree formed by a pair and its elements* are the same. This intuiton suggests the folowing recursive definition of the notion *type* over structural field elements:

1. *All objects are type-identical with themselves;* (S2-9)
2. *Distinct pairs are type-identical if and only if their* CARs *and* CDRs *are (recursively) type-identical;*
3. *No other distinct s-expressions are type-identical;*

A problem with S2-9, however, is that it leaves undefined the question of whether certain circular structures are type-identical. The problem is that a the CAR and CDR relationship on any given pair may yield an arbitrary graph, not a tree. In particular, the second clause in definition S2-9 is ill-defined where each of two distinct pairs P_1 and P_2 is its own CAR and CDR (many other simple examples are possible). A better characterisation is the following (by *type-distinct* we mean *not type-identical*), which maintains the intuition that distinct leaves indicate distinct types:

1. *All objects are type-identical with themselves;* (S2-10)
2. *No atom or numeral is type-identical with any object other than itself;*
3. *Two pairs are type-distinct if either their* CARs *or their* CDRs *are type-distinct.*
4. *Any two pairs which are not shown to be type-distinct by rules 1 — 3 are type-identical;*

This definition will play a role in the definition of type-identical *lists* in the following section. It is slightly coarser in grain than one might at first suspect, in that it sets the P_1 of the previous paragraph as type-identical with a structure consisting of two dinstinct pairs P_3 and P_4, each of which is the other's CAR and CDR. Revisions of S2-10 are possible that establish finer-grained equivalence classes of structures, so as to distinguish the example just given. However S2-10 will serve our purposes.

In spite of this definition of a *structural* (as opposed to a *notational*) notion of type, we will remark explicitly when we are using the term "type" with respect to structural field

objects; its primary meaning will remain a notion defined over lexical notations.

As well as the primitive notions of numeral, atom, and pair, there is a derived notion of *list* — over which procedural consequence is most naturally defined. In actual use lists, rather than pairs, are by far the more commonly used structural abstraction. A simple notion of a list can be inductively defined as follows: a ***list*** is either:

1. *the distinguished atom* NIL, *or* (S2-11)
2. *a pair whose* CDR *is a list.*

The ***length*** of a list L is said to be 0 if L is NIL, or else 1 greater than the length of its cdr. A list has as many ***elements*** as its length: we will say that its first element is its CAR, and its Nth element is the <N-1>th element of the list that is its CDR.

A number of properties of lists follows from this characterisation. First, it is not necessary that the elements of a list be atoms, numerals, or lists: they may be non-list pairs. Second, there is a non-isomorphism between *pairs* and *lists*: all lists but one (the empty list NIL) are pairs, but not all pairs are lists. Third, although the definition as given does not admit lists of infinite length, nothing excludes a list from being one of its own elements, just as a pair can be its own CAR.

There are two problems with S2-11 which need attention. First, on this account a list is not a composite object containing its elements, unless "containing" is defined to include the transitive closure of the CDR relationship. It follows that on this view one could change an element of a list without changing the list itself, since the list is identified with a single "head" pair, which by our prior account of identity and mutability is not thereby changed. This is a mildly unhappy terminological consequence. An obvious way to revise the definition would be to define a list to be an abstract *sequence* of pairs, each of which was the CDR of the previous pair: in this way if one changed some element of a list (changed the CAR-relationship of one of the pairs in the chain) one would on the theoretical account have a different list (we can't *absorb* the notion of change directly in a mathematical account, since mathematical entities are not subject to modification). However arguing against this revision is the consequence that a list would then no longer be an element of the structural field: a list could not, for example, be the CAR of some other pair.

What we are up against, of course, is the fact that lists are in essence an abstract data structure *implemented* as chains of pairs in the 1-LISP field. Characterising them in terms of their implementation is too detailed to be aestheticly satisfactory, even though this is virtually the only implementation widely utilised (although others — particularly ones with different temporal properties — are occasionally explored). Furthermore, it seems to lead to the awkward use of prior terminology. On the other hand, characterising them abstractly seems to take us out of the LISP field in ways that our present conceptual vocabulary is not equipped to handle. At the end of chapter 3 we will examine data abstraction explicitly; until that time we will accept the identification of a list with its head pair (or NIL), since that introduces fewer formal difficulties.

The other problem with S2-11 is that it excludes certain arrangements of pairs that we will want to consider *circular* lists. As opposed to the foregoing difficulty, this trouble can be accomodated in a revised definition. Informally, we would prefer to define a list as either NIL or as a pair whose transitive closure of CDR's included no numerals or atoms (other than NIL). As was the case with type-identical pairs, the solution is to explicitly exclude all non-lists, and then to define the lists to be the complement of this set. This approach can be effected as follows:

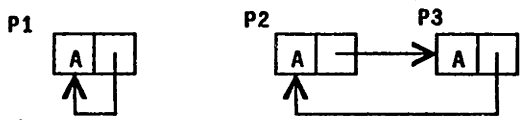
1. *The atom* NIL *is a list;* (S2-12)
2. *No other atom or numeral is a list;*
3. *If the* CDR *of a pair is not a list, the pair is not a list;*
4. *All other pairs are lists.*

The definition of length given above can be retained for finite lists; if the transitive closure of the CDR relationship of a list pair does not terminate with NIL in a finite number of steps, we will simply posit that the length of the list is infinite. Thus two sorts of structural arrangements might be lists of infinite length: those consisting of an infinite number of pairs, and those comprising a finite number of pairs where one of those pairs is the CDR of a pair in the transitive closure of its own CDR relationship (such as the P_1 mentioned earlier).

Since we have continued to identify lists with pairs, the definition of *type-identical* given in S2-10 applies directly to lists, with the consequence that all infinite-length lists with the same elements are type-identical, even though, as mentioned above, there is a natural sense in which some of them can be distinguished. For example, suppose that pair

P_1 has the atom A as its CAR and is its own CDR, and that pairs P_2 and P_3 each have the atom A as their CAR and are each other's CDR. Using the graphical notation to be introduced in the next section, these structures would be notated as follows:

(S2-13)



On the account we have adopted P_1 and P_2, though distinct, are type-identical (and both are type-identical to P_3), and are of equal (infinite) length. As we mentioned above, it is possible to adopt a finer-grained type-identicality predicate to distinguish such cases, but we will not need to do that here.

It is with reference to *lists*, and not to *pairs*, that many aspects of both interpretive consequence and declarative import will be defined, in part because the lexical notation is a more natural notation for lists than for arbitary pairs, as we will show in the next section. As we have time and again remarked, such categorical ambiguity will make it very difficult to align the double semantical accounts we will in the end adopt. For present purposes, however, since we are dealing only with a behavioural specification of interpretive consequence, this notion of list will serve.

This completes the account of 1-LISP's structural field. We have of course dealt with it purely as an abstract collection of formal structure: neither notation, procedural consequence, nor semantical import have yet been mentioned (and thus we are not yet in a position to raise any semantical queries). In addition, we have so far discussed primarily categorical structure: the only *individual* to play a role in describing 1-LISP's field is the distinguished atom NIL, used to define the derived notion of a list. In addition, we are accepting the notion of an abstract virtual machine: the definition of the 1-LISP field makes no comment on how 1-LISP is implemented. Thus no notion of pointer will intrude on our discussion, nor will the creation of atoms, or garbage collection.

From these definitions it follows that a number of typically-available features are missing in 1-LISP, such any access to all atoms (the LISP *oblist*), etc. In addition, as we will describe in subsequent sections, atoms are used in 1-LISP programs as identifiers and variables, and an association between them and their *values* (which are always elements of the field) is maintained. This association, however, is part of the state of the processor, and

as such is not encoded within the field itself. Thus we have not identified a "value" mapping over the atoms, nor will we store atom values under a "value" property of an atom's property list. When we design 3-LISP we will have to have *environment designators* (structures that designate such assocations) available as full-fledged structural objects, but until that time the information about atoms and their values is considered to remain a part of the internal state of the processor, not a manifest aspect of the structural field.

The basic character of the 1-LISP structural field just outlined will be mainly preserved in subsequent dialects — atoms, numerals, and pairs, in particular, will remain unchanged. In 2- and 3-LISP we will introduce a primitive syntactic type called a *rail* to serve in place of 1-LISP's derived notion of *list*, we will introduce two separate boolean constants that are not atoms, and we will deal with quoted forms specially. But the locality considerations outlined above will remain the same for the categories that are preserved, and similar metrics will be introduced on the new types (rails, for example, will receive the asymmetric accessibility relations of lists). The notational interpretation function Θ_G (see chapter 3) will be modified, and of course both declarative and procedural semantics will be adjusted. However all these modifications will be defined as changes with respect to this 1-LISP field; characteristics that are not again mentioned should be assumed to carry through intact.

We can model the 1-LISP structural field as follows. First, we define three sets ***PAIRS***, ***ATOMS***, and ***NUMERALS*** of pairs, atoms, and numerals, and three relationships to model ***CAR***, ***CDR***, and ***PROP***. In the earlier discussion we said that the ***PROP*** relationship took atoms to pairs, but here we have corrected that so that it maps atoms onto lists. Note as well that whereas ***S*** is a fixed set, the set ***FIELDS*** is a *set* of fields, intended to include all possible *states* of the 1-LISP field; thus any given state of the field is modelled as an element of ***FIELDS***. The reason is that we define ***CARS*** to be the full *set* of CAR relationships, intended to model *changing* CAR relationship, and so forth. This approach is an instance of a standard meta-theoretical manoeuvre to compensate for the fact that change cannot be absorbed into mathematics.

PAIRS	≡ { P | P is a pair }	— the set of pairs	(S2-14)
ATOMS	≡ { A | A is an atom }	— the set of atoms	
NUMERALS	≡ { N | N is a numeral }	— the set of numerals	

$$\begin{array}{lll} \mathbf{S} & \equiv \mathbf{PAIRS} \cup \mathbf{ATOMS} \cup \mathbf{NUMERALS} & \text{— the structural field elements} \\ \\ \mathbf{PROPS} & \equiv [\mathbf{ATOMS} \rightarrow \mathbf{LISTS}] & \text{— the "property-list" relationship} \\ \mathbf{CARS} & \equiv [\mathbf{PAIRS} \rightarrow \mathbf{S}] & \text{— the "CAR" relationship} \\ \mathbf{CDRS} & \equiv [\mathbf{PAIRS} \rightarrow \mathbf{S}] & \text{— the "CDR" relationship} \\ \\ \mathbf{FIELDS} & \equiv \mathbf{S} \times \mathbf{PROPS} \times \mathbf{CARS} \times \mathbf{CDRS} & \text{— the set of structural fields} \end{array}$$

Furthermore, we will define three meta-theoretic functions CAR, CDR, and PROP, that take an element of $\mathbf{S}$ and a field, and yield that corresponding element of $\mathbf{S}$ in that field:

$$\text{CAR} : [[\mathbf{F} \times \mathbf{PAIRS}] \rightarrow \mathbf{S}] \qquad \text{(S2-15)}$$
$$\equiv \lambda F, \lambda P \,.\, F^{3}(P)$$

$$\text{CDR} : [[\mathbf{F} \times \mathbf{PAIRS}] \rightarrow \mathbf{S}] \qquad \text{(S2-16)}$$
$$\equiv \lambda F, \lambda P \,.\, F^{4}(P)$$

$$\text{PROP} : [[\mathbf{F} \times \mathbf{ATOMS}] \rightarrow \mathbf{LISTS}] \qquad \text{(S2-17)}$$
$$\equiv \lambda F, \lambda A \,.\, F^{2}(A)$$

We will let variables P, P_1, P_2, P', etc., range over ***PAIRS***, A, A_1, A_2, A', etc., range over ***ATOMS***, S, S_1, S_2, S', etc., range over all elements of ***S***, and so forth, both for explicit quantification and for lambda abstraction.

There are some identity interactions between our English characterisations of the 1-LISP field and these mathematical constructs, deriving from the fact that a structural field is of course not actually an ordered pair of sets and functions; it can merely be *modelled*, at any given moment, with such a mathematical abstraction. Modelling is itself a semantical operation, and yet another intepretation function relates the domain being modelled with the model; one of the questions that an account of such a mapping would have to answer is that of how object identity in the source domain is modelled in the target model. We have spoken of the 1-LISP field *changing* from time to time; in our mathematics, since no mathematical entity can change, we model each change with a new mathematical object. Thus the set ***FIELDS*** models the set of all possible states of a field; each individual state is modelled with an element of that set. In the mathematical characterisation of an operation (like RPLACA) that changes the field, we will describe the state of the field with a new elements of ***FIELDS***.

With respect to these definitions we can define the set ***LISTS*** of lists (given a field), as suggested earlier. Our first attempt, in which lists were those pairs whose CDR's were lists

or the atom NIL would be recursively defined as follows:

$$\textit{LISTS} \equiv \lambda F \in \textit{FIELDS}\ [\{\text{NIL}\} \cup \{P \in \textit{PAIRS} \mid \text{CDR}(P,F) \in \textit{LISTS}(F)\}] \qquad \text{(S2-18)}$$

This characterisation, as we noted, was unacceptable in ignoring non-tree lists. The second suggestion, in which we identified lists with *sequences* of their elements, would be modelled as follows:

$$\begin{aligned}\textit{LISTS} \equiv\ & \lambda F \in \textit{FIELDS} \qquad \text{(S2-19)}\\ & [\{\langle\rangle\} \cup\\ & \ \{\langle S_1\ S_2\ \ldots\ S_k\rangle\\ & \quad [\![\ \forall i\ 1 \le i \le k\\ & \qquad [\text{CDR}(S_i,F)=S_{i+1}] \wedge\\ & \qquad [\![\ \text{CDR}(S_k,F) = \text{NIL}\ ] \vee [\ \text{CDR}(S_k,F) = S_i,\ 1 \le i \le k\ ]]]\}\end{aligned}$$

As mentioned in the discussion, however, this too had a number of unacceptable consequences, including the fact that it removed lists from the structural field. The definition we settled on, sketched in S2-12, can be mathematically modelled as follows:

$$\begin{aligned}\textit{NON-LISTS} \equiv\ & \lambda F \in \textit{FIELDS}\ [\textit{ATOMS} \cup \textit{NUMERALS} - \{\text{NIL}\} \qquad \text{(S2-20)}\\ & \qquad \cup \{P \in \textit{PAIRS} \mid \text{CDR}(P,F) \in \textit{NON-LISTS}(F)\}]\end{aligned}$$

$$\textit{LISTS} \equiv \lambda F \in \textit{FIELDS}\ [S - \textit{NON-LISTS}(F)]$$

We can also define the *type-identity* predicate outlined in S2-10. A first suggestion is:

$$\begin{aligned}&\text{TYPE-EQUAL} : [[F \times S \times S] \to \{\text{TRUE, FALSE}\}] \qquad \text{(S2-21)}\\ &\quad \equiv \lambda F \in \textit{FIELDS},\ S_1,\ S_2 \in S\\ &\qquad [\textit{if}\ [S_1 = S_2]\\ &\qquad\quad \textit{then}\ \text{TRUE}\\ &\qquad\quad \textit{elseif}\ [[S_1 \notin \textit{PAIRS}] \vee [S_2 \notin \textit{PAIRS}]]\\ &\qquad\qquad \textit{then}\ \text{FALSE}\\ &\qquad\qquad \textit{else}\ [[\text{TYPE-EQUAL}(F,\text{CDR}(S_1,F),\text{CDR}(S_2,F))] \vee\\ &\qquad\qquad\quad [\text{TYPE-EQUAL}(F,\text{CAR}(S_1,F),\text{CAR}(S_2,F))\ ]]]\end{aligned}$$

However this is too *computational* an attempt to avoid infinite regress, and is undefined on just those cases we took pains to include: circular structures. A better approach is, for each element, to identify those structures to which it is type-distinct, and then to define type-identity with reference to this set:

$$\begin{aligned}&\text{DISTINCTS} : [[F \times S] \to S^{\bullet}] \qquad \text{(S2-22)}\\ &\quad \equiv \lambda F \in \textit{FIELDS},\ S \in S\\ &\qquad \textit{if}\ [S \in \textit{ATOMS} \vee S \in \textit{NUMERALS}]\ \textit{then}\ S - \{S\}\\ &\qquad \textit{else}\quad \textit{ATOMS} \cup \textit{NUMERALS} \cup\end{aligned}$$

$$\{P \in \textit{PAIRS} \mid \text{CDR}(P,F) \in \text{DISTINCTS}(F,\text{CDR}(S,F))\} \cup$$
$$\{P \in \textit{PAIRS} \mid \text{CAR}(P,F) \in \text{DISTINCTS}(F,\text{CAR}(S,F))\}$$

$$\text{TYPE-EQUAL} : [[F \times S \times S] \rightarrow \{\text{TRUE}, \text{FALSE}\}] \equiv \lambda S_1, S_2\ [S_1 \notin \text{DISTINCTS}(F,S_2)] \quad \text{(S2-23)}$$

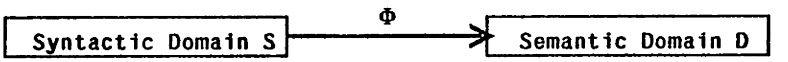
From this definition the appropriate symmetry relationship can be proved:

$$\forall F \in \textit{FIELDS}, S_1, S_2 \in S\ [\text{TYPE-EQUAL}(F,S_1,S_2) \equiv \text{TYPE-EQUAL}(F,S_2,S_1)] \quad \text{(S2-24)}$$

It is clear that the structural field as defined is approximately a graph, consisting of three node types (atom, numeral, and pair) and three asymmetric labelled arcs (CAR, CDR, and property-list), with certain restrictions on the types of arcs. In addition there are a handful of distinguished nodes. This characterisation will be of some use in subsequent proofs. However we also have defined a locality or accessibility relationship over the elements (nodes) of *S*, which forms a different but related graph: the accessibility relationship is a directed arc between nodes as well, but it is a different arc from the other three — it is different in *kind*, rather than being a fourth variety. If *FIELDS* were defined to be a graph of the first sort, then one could define a related graph *FIELDS** consisting of the same nodes, with the accessibility relationship as the directed arc, rather than the three primary binary relationships. Of course *FIELDS** would be highly dependend on *FIELDS*, since the accessibility relationship must correspond topologically to a subset of the primary relationships. A *general* definition of a structural field could be found in this direction, but such general goals are not our present task. We will talk more simply and informally in terms of the particular structural fields we will define.

Chapter 3. Semantic Rationalisation

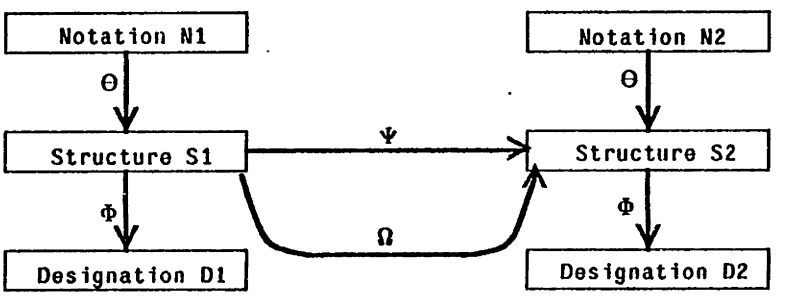
Our next task is to subject 1-LISP to semantical scrutiny, with the hope of clarifying the assumptions and principles that underlie its design. A general introduction to our approach was given in section 1.d; we will repeat here for reference four diagrams that summarise our main terminology. First, we said that in general we take a *semantical interpretation function* to relate elements of a syntactic domain with corresponding elements of a semantic domain, as follows (here Φ is the interpretation function from syntactic domain S to semantic domain D):



(S3-1)

We then presented the following more complex version of this diagram, intended to cover the general computational circumstance, where Θ is the interpretation function mapping notations into elements of the structural field, Φ is the interpretation function making explicit our attributed semantics to structural field elements, and Ψ is the function formally computed by the language processor.

(S3-2)



With respect to this diagram, we said that we would prove the following *evaluation theorem* for 1-LISP (and therefore by implication for all standard LISPs, including SCHEME):

$$\forall S \in \mathcal{S}\ [\,\mathit{if}\ [\Phi(S) \in \mathcal{S}]\ \mathit{then}\ [\Psi(S) = \Phi(S)]\ \mathit{else}\ [\Phi(\Psi(S)) = \Phi(S)]\,] \qquad \text{(S3-3)}$$

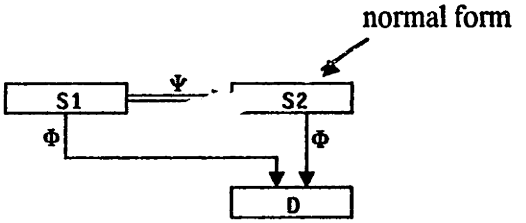
In contrast, we are committed to the construction of a dialect satisfying the following equation (the *normalisation property*) — much more similar to the procedural regimens defined over classical calculi (logic, the λ-calculus, and so forth, as we will show in this

chapter):

$$\forall S \in \mathcal{S}\ [[\Phi(S) = \Phi(\Psi(S))] \wedge \text{NORMAL-FORM}(\Psi(S))] \qquad \text{(S3-4)}$$

The procedural regime that it describes can be pictured as follows:

(S3-5)



In this present chapter we will investigate these semantical issues in detail, beginning with an analysis of the semantical analysis of traditional systems, turning then to a consideration of semantics in a computational setting, and then taking up the task of setting out a full account of the semantics, both declarative and procedural, of our basis 1-LISP dialect.

3.a. The Semantics of Traditional Formal Systems

Diagram S3-2 is sufficiently general that we can characterise a variety of traditional formal systems in its terms, beginning with logic and standard model theory.

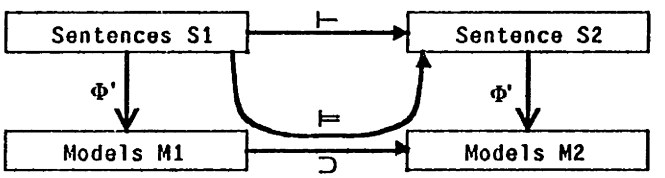
3.a.i. Logic

When formalising the (declarative) semantics of, say, a first-order language, one lays out assumptions about the denotational import of the various predicate letters and terms, and then identifies (usually making use of the recursive compositionality of the grammar) the semantical import of composite expressions as a function of the ingredients. Thus we might say that the letter Q, R, and S designate one-place predicates, the atomic terms A, B, and C designate objects in the domain, that sentences of the form P(X) are true (designate Truth, to be Fregean) just in case the predicate designated by the predicate letter P is true of the object designated by the term X, and so forth. Similarly we might add that sentences of the form P $\wedge$ Q are true just in case P is true and Q is true, that sentences of the form P $\supset$ Q are true just in case P is false or Q is true, and so on.

Independent of this semantical account one defines an inferential regime that maps sentences or sets of sentences onto other sentences. Such an inferential regime is defined to obey what are called *inference rules* that state which transformations are legal. *Modus ponens*, for example, is a rule that, given sentences P and P $\supset$ Q, would yield Q. Crucially, the inference rules are defined over the *form* of the expressions involved — not with reference to the sematical weight they are taken to bear. What one then attempts to prove, typically — and this is the point — is that this inference rule is *sound*, which is to say, that in all cases the sentence that it yields will be true in those cases in which the sentences it uses are true. If the conclusion (which we may call Y) *semantically follows from* the assumptions (X), we write X $\models$ Y; if the inference regime will produce Y given X we write X $\vdash$ Y. To say of an inference regime that it is sound is to say that X $\vdash$ Y only if X $\models$ Y. To say that it is *complete* is to say that if X $\models$ Y, then X $\vdash$ Y. Only after one has established consistency and completeness (possible only in some languages, and of course proved impossible for all logics with the power of arithmetic) can one treat the *entailment* relationship "$\models$" and the *derivability* relationship "$\vdash$" as equivalent, *in an extensional sense:*

they relate the same sentences. They are of course different in *meaning*: saying that $S_1 \vdash S_2$ is different from saying that $S_1 \models S_2$; it is in fact exactly because they are different that it is powerful to show that they are extensionally the same. In particular, it is crucial to realise that entailment is fundamentally semantical; derivability fundamentally formal.

These few points are illustrated in the following diagram. By Φ' (what in the philosophy of language is called a *satisfaction* relationship) we refer to a relationship between sentences and models in which that sentence is true; this is the standard way in which entailment is defined. Thus S1 $\models$ S2 just in case Φ'(S1) $\subset$ Φ'(S2) — that is, just in case S2 is true in all models in which S1 is true. For example, if S1 is the sentence [$\forall$X MONTH(X) $\supset$ DAYS-IN(X, 30)] $\wedge$ [MONTH(FEBRUARY)] and S2 is the sentence DAYS-IN(FEBRUARY, 30), then S2 is entailed by S1 because in all models in which February is a month and all months have 30 days, February is of 30 days duration as well.



(S3-6)

Entailment ($\models$) is not a relationship between *models*; like derivability ($\vdash$) it is a relationship betwen sentences. Being semantical, however, $\models$ in a sense "reaches down through the semantics" of the sentences involved. In contrast, $\vdash$ is a purely formal relationship that holds between sentences solely in virtue of their grammatical structure. What is crucial about $\models$ is that it be definable purely in terms of S1, S2, and Φ alone; the definition of $\models$ must not rest on the definition of $\vdash$, or on any of the machinery defined to implement it.

The satisfaction relationship Φ, in other words, and the derivability relationship $\vdash$, must be *independently* definable.

All this is of course well-known — we have reviewed it to set our understanding of LISP up against it for comparison. With respect to such a comparison the following points are relevant: there are *three* relationships of interest mentioned in the preceding discussion. The first is the relationship between symbols and their designations (analogous to what we will call Φ); the second is a formally-definable relationship between expressions ($\vdash$); the third is a semantical relationship between sentences ($\models$) that depends on their designations.

The proof of correctness of the formal relationship $\vdash$ is that it correctly mimics $\models$. $\vdash$ *does not correctly mimic* Φ. Φ is necessary in order to *define* $\models$, but Φ *is not itself* $\models$. Our criticism of standard programming language semantics, at least for languages like LISP perfused with pre-computational semantical attribution, will be that the LISP analogues of Φ and $\models$ are unhelpfully conflated.

The comparison between S3-6 and S3-2 is clear: in logic, there is no distinction made between notation and abstract structure; the inference rules and the semantics are defined with respect to *sentences* (sentence *types*, to be precise, but S in S3-2 is not simply the type of N), not with respect to an abstract structure or field into which sentences are translated. Thus the N1 and S1 of figure S3-2 are coalesced into S1 in S3-6. The relationships between the other elements of the figure, however are these: satisfaction is logic's Φ; derivability ($\vdash$) is logic's Ψ, and entailment ($\models$) is logic's Ω.

Two further points are notable regarding logic's Ψ and Ω — the derivability and entailment relationships. First, they are not *functions*: from any given sentence or set of sentences there are an infinite number of other sentences that can be derived; there are an infinite number of other sentences that are entailed. In our deterministic computational formalisms, in contrast, we will of course have to define a more narrowly constrained Ψ that is at least approximately a function.

Second, as mentioned above, the entailment relationship is defined without reference to the derivability relationship. Since entailment is not a function, however, we do not have a situation in which, for any given sentence, entailment takes it to a particular sentence, and derivability takes it to a particular sentence, and then the proofs of soundness and completeness show that this is the same sentence. We do not, to put this more formally, have the following equation saying that two *expressions* are the same (i.e. the following equation is semantically ill-formed, because Ω and Ψ are not functions):

$$\forall S_1\ \exists S_2\ [[\, S_2 = \Omega(S_1)\ ] \supset [\, S_2 = \Psi(S_1)\,]] \qquad \text{; False for logic} \qquad \text{(S3-7)}$$

Instead, in a sound and complete logic one instead proves the following, which says that two *sets of expressions* are the same:

$$\forall S\ [\{\, X \mid S\ \Omega\ X\,\} = \{\, X \mid S\ \Psi\ X\,\}] \qquad \text{(S3-8)}$$

Using the more familiar labels peculiar to logic, this latter equation can be rewritten as (in fact there are subtleties: the statement [S ⊨ X ≡ S ⊢ X] is stronger than [⊨ X ≡ ⊢ X], for example if S is infinite, implying that { X | S ⊨ X } may be larger than { X | S ⊢ X }, but the intent is clear):

$$\forall S\ [\{X \mid S \models X\} = \{X \mid S \vdash X\}] \qquad \text{(S3-9)}$$

In designing 2- and 3-LISP we will have the stronger equation S3-7 as our goal, although we too will fail to reach it — will fail to attain a computable version of Ω that is a function.

3.a.ii. The λ-Calculus

In the λ-calculus, as in logic, standard denotational methods are used to describe possible *models* of λ-calculus expressions. The *reduction regimes* defined over such expressions (α- and β-reduction, typically) are then shown to be sound and complete, in the following sense: every expression B to which an expression A reduces can be shown to have the same designation as A. *The λ-calculus's Ψ, in other words, is always designation preserving.* For example, suppose we have the expression

```
[λZ. ((λY. (λZ.YZ)) Z)] (λG.G)                                    (S3-10)
```

Then by a series of reductions we would be given the following derivation:

```
[λZ. ((λY. (λZ.YZ)) Z)] (λG.G)                                    (S3-11)
[λZ. ((λY. (λW.YW)) Z)] (λG.G)          ; α-reduction
[λZ. (λW.ZW)] (λG.G)                    ; β-reduction
(λW. (λG.G)W)                           ; β-reduction
λW.W                                    ; β-reduction
```

The last line designates the identity function in the standard model, which is to say, *is mapped by the standard interpretation function Φ onto the identity function.* What is true, therefore, is that *each* of the lines of S3-11 designates the same function, since neither α-reduction nor β-reduction changes designation. In addition, the last line is in *normal form*, which is defined in the λ-calculus as being an expression to which no further β-reduction rule applies.

In the λ-calculus, in other words, Φ is the interpretation function, and Ψ is the transitive closure of α- and β-reduction. That Ψ is approximately a *function*, up to α-interconvertability, is proved in the Church-Rosser theorem: although at any given stage in the reduction of a lambda-calculus expression there may be more than one option of how

to apply an α or β reduction rule, if an expression S reduces to normal form via one path of reduction steps, it will reduce to that normal form expression, or one convertible to it via α-reductions alone, via any other path of reduction steps. There are subtleties, such as that some well-formed expressions do not reduce to a normal form, but the permeating character of the λ-calculus's reduction scheme is that expressions are taken by Ψ towards a co-designating expressions that are not further reducible.

There are various options open in defining an interpretation function Φ for the λ-calculus: although in what we will call *standard* models each lambda term designates a function, other possibilities are sometimes chosen. Various proofs of consistency, for example, select as the designation of a term E the class of all sentences of the lambda-calculus interconvertible with E by α- and β-reduction (it is thus immediate that α- and β-reduction are designation preserving: this is one of this model's great conveniences). Nonetheless we will assume the standard model, where λ-terms designate standard (continuous) functions, throughout our discussion. (Certainly no LISP aficianado can easily see λX.X as designating anything but the identity function.)

The λ-calculus differs from logic in two important ways: there is a much stronger sense that its Ψ takes expressions towards a definite goal (a non-reducible term) than is the case in logic, where Ψ ($\vdash$) leads to an infinite set. In a certain sense, in other words, the λ-calculus's Ψ is *stronger* than is logic's Ψ. On the other hand, the λ-calculus does not have a particularly well-specified Ω: the concept of normal-form is defined with respect to the lack of further applicability of the inferential protocols, not independently to any salient degree. One could argue that this Ω is no weaker than logic's Ω (entailment), but what is different is that in logic Ψ and Ω ($\vdash$ and $\models$) are *equivalent* in restrictiveness: in the λ-calculus Ψ is much more finely specified than is the Ω that makes no reference to the reduction scheme (it merely says that designation is preserved).

In contrast, we will require of 2-LISP that the definition of Ω be complete — at least up to the identification of category, and, for all but function designators, up to type-equivalence — prior to the definition of Ψ. Our notion of normal form, in addition, will be different from that used in the λ-calculus: we will define normal-form primarily in terms of the type of the referent (D1, and equivalently D2, in S3-2), and partially in terms of the form of the original designator (S1 in S3-2): no reference will be allowed to the mechanisms

that transform that original expression.

A comment in passing. The reader will note that we are defining for our own purposes a variety of traditional technical terms, of which "normal form" is a good example. Every writer faces the question of whether familiar or new terms will best convey a new understanding, frequently adopting some mixture. We too will introduce some new terminology, but will also stretch some familiar terms to fit our circumstances, particularly when the essence or fundamental insight on which the original notion was based seems also to lie at the heart our new idea. Thus by "normal form", for example, we signify a concept related, but not identical, to the notion of the same name used in the λ-calculus; it is our sense, however, that the essential qualities of a λ-calculus normal form expression — being stable under processing, context independent, and in some informal sense minimal — are preserved in our extended notion. In aid of the reader, however, we will make every effort to note explicitly any circumstance in which we use a traditional term with other than its received meaning.

3.a.iii. PROLOG

It is instructive to look next at PROLOG,[1] as our first example of a computational formalism, since PROLOG is widely advertised as semantically strict, and derivative from logic. PROLOG, as mentioned in the introduction, is at heart a *dual-calculus* formalism, in that the procedural consequence and declarative import are signified by what amount to different languages super-imposed one upon the other. One of these languages — the one over which declarative semantics is defined — is a subset (Horn clauses) of the first order quantificational logic: the declarative interpretation function is then inherited from logic directly. Φ for PROLOG, in other words, is Φ from first-order logic, without modification.

It follows, then, that PROLOG's Φ is not based on *computational* considerations, since logic is not a computational system. PROLOG also has a formally-defined relationship among sentences (among processor and field states, properly, but we are being informal for the time being) computed by the PROLOG processor: this is PROLOG's Ψ. Because it inherits Φ from logic, standard notions of entailment (⊨) are defined; one can then prove that PROLOG's Ψ implements a subset of logic's ⊨. Since entailment is not a *function*, one does not prove that Ψ *correctly embodies* ⊨; rather, the PROLOG designers have proved that Ψ

embodies a subset of $\models$ (by showing that Ψ implements a subset of a provably sound $\vdash$).

Thus PROLOG has a cleanliness that LISP lacks: Φ and Ψ are independently defined. This allows a proof that the PROLOG processor is correct (it embodies a subset of $\models$) — something that cannot be done for LISP. For we have no *prior* notion of what LISP should do: we can therefore prove correct only *implementations* of LISP, or *programs that designate the LISP evaluator*, or *meta-linguistic characterisations of evaluation*, and so forth. In Gordon,[2] for example, we find a proof that the meta-circular definition of EVAL as given in the LISP 1.5 manual is correct: by this is meant that, given a meta-theoretic definition of LISP evaluation, the evaluation of the definition of EVAL will yield behaviour equivalent to that of direct evaluation. But this is not a proof that LISP evaluation is *itself* correct, in any sense, because there is no pre-computational intuition as to *what LISP evaluation should be.* By analogy, if I asked you whether a device that I built in my backyard was correct, you would have to ask me what it was supposed to do, before my question would make sense. If my reply was only that it is designed to manifest its own behaviour, then my original question is rendered circular.

In contrast to LISP, PROLOG is defined in terms of a pre-computational characterisation of what its processor is trying to do: it is trying to maintain truth, in terms of an independently specified truth theory (model theory for first logic). Thus, in this limited sense — limited because it does not deal with what subset of entailment the PROLOG processor computes, or about side effects and so forth — it *is* meaningful to ask whether PROLOG's Ψ is correct.

After spelling out the declarative semantics naturally attributed to LISP structures, and sketching the architecture of a rationalised design, we too, like the PROLOG designers, will be able to ask whether a proposed LISP processor is correct. In chapters 4 and 5 it will be required of us to demonstrate that this question, for 2-LISP and 3-LISP, can be affirmatively answered.

3.a.iv. Commonalities

The crucial facts that permeate the discussions of the foregoing three systems (logic, the λ-calculus, and PROLOG) are three:

1. Semantical import was attributed to the expressions or structures of each formalism *prior* to the definition of a procedural regime over those expressions or structures.
2. The procedural treatment was defined independently of the semantic attribution. This is the direct manifestation of the fact that logic, the λ-calculus, and PROLOG are *formal* systems: how things go is defined in terms of formal structure, not semantical weight.
3. The procedural function Ψ was related to the attributed semantical interpretation function Φ in a particular way: Ψ was Φ-*preserving*, mapping expressions onto other expressions with the same designation (or, in logic's case, onto expression with more inclusive designations).

Points 1 and 2 establish that semantical weight and procedural treatment are independently specified. It is only because of this independently attributed semantics that the procedural protocols could be semantically characterised: if it were not for the prior existence of Φ, the relationship Ψ would simply be any relationship at all, and Ω would not exist.

In contrast with such similarities among these three systems, we have noted that LISP systems are not traditionally analysed in this manner. Evaluation *is* the procedural treatment: the import of LISP constructs is characterised in terms of the procedural consequence (the common wisdom that LISP's QUOTE is an operator *that defers one level of evaluation* is a classic example). Thus no true semantical analysis of evaluation is possible under the standard analysis.

In constrast with tradition, we have said that 2-LISP's and 3-LISP's procedural regimes will be based on a *normalising* processor: that the Ψ of those dialects will take structures into normal-form codesignators. It should by this point be evident that to define a normalising dialect *presupposes* what we are calling a double semantics: that the notion of normalisation and co-designation makes sense only when a declarative semantics is formulated prior to and independent of the procedural treatment. It is for this reason that laying out the natural declarative semantics of LISP is a prerequisite to defining 2-LISP. Defining a normalising or simplifying dialect of LISP, in other words, is not straightforward: it requires the explicit formulation of an entire theory of semantics for LISP structures that is prior to and independent of any account of how the LISP processor is to function. This is the bottom line of this entire sketch of semantics as traditionally construed.

One final comment deserves to be made regarding traditional procedural treatments, stated in the third point listed above. We have pointed out that reduction in the λ-calculus

and derivability and proof procedures in logical systems are Φ-preserving. So too are mathematical simplification rules over algebraic and arithmetic expressions. It is not unnatural to ask, especially if one is primarily familiar with LISP, why Φ-preservation is so common a semantical trait of procedural regimens. Nothing in points 1 or 2 above requires that Ψ bear *this* particular relationship to Φ: all that they require is that Φ and Ψ be independent. Furthermore, aesthetic considerations merely imply that *some* coherent relationship between the two be honoured: Φ-preservation is presumably just one of any number of alternatives (that Ψ and Φ be *identical* would be even simpler, for example).

There are two parts to this answer to this query. First, the great bulk of language speaks about the world, rather than about other language. We communicate, primarily, about some subject matter: the shift into talking about our communication is a less natural, and considerably more problematic, matter than simple linguistic behaviour that "stays at a given semantical level". Level crossing behaviours of all kinds — from simple use/mention shifts to full reflection — is, as this dissertation is of course at pains to make clear, a valid and coherent subject matter of independent interest. Our concern with it, however, must not mislead us into thinking that anything other than simple, constant-level symbolic behaviour constitutes the vast majority of linguistic and formal practice.

Secondly, "about-ness" is exactly what the formally-defined notion of designation is intended to capture. As we have constantly said, designation cannot be defined in arbitrary ways precisely because of this point, and our formal attempts to define the notion succeed just to the extent that they rationally reconstruct lay intuitions. In addition, about-ness must be faced if we are to construct a reflective architecture, because the defining quality of reflection is that one's thoughts are *about* one's own thought processes. Thus in order to show that, when it reflects, the programs that 3-LISP runs are about its own operations and structures, we will have to make reference to the designation of 3-LISP terms.

Φ-preserving behaviour, in other words, is by far the most natural kind, and it is straightforward that artificial formal systems should be defined in this way. It must be admitted in addition, however, that there is no great temptation to define the three systems we have just considered in any other way, since anything other than Φ-preservation would be impossible. For example, on the standard interpretation, λ-calculus expressions designate infinite functions, not other expressions, and there is simply no *possibility* of

having the syntactic transformation function be a de-referencing function. In programming languages, however, when we concentrate on that portion of the structural field embedded in *programs*, we deal almost exclusively with terms whose referents are other syntactic expressions. This is not merely the case with such complex facilities as LISP's macros, FEXPRs, and the like — those deal with terms whose referents are other pieces of *program* structure. But virtually *all* terms in programs other than function and mathematical designators deal with *data* structures, which are themselves syntactic. It is the introduction of such terms — and the concomitant embedding of the syntactic domain within the semantic domain — that has apparently led to a temptation on the part of the formalism designers to make the formally defined expression processor (Ψ) de-reference those expressions for which de-referencing is possible (remains with the syntactic domain S), as is indicated in S3-3 above. It is our mandate to admit and welcome these meta-structural levels of designation, while preserving the basic co-designation processing that characterises these simpler systems. We adopt this mandate in part because of our recognition that explicit level-crossing and reflective behaviours are by far and away most easily introduced into a system that by default preserves designation — into a system, we will say, that by default remains ***semantically flat***.

3.b. The Semantics of Computational Calculi

Showing that the evaluation theorem holds for 1-LISP, and arguing for the increased clarity of a rationalised dialect, are straightforward tasks, once the interpretation functions Φ and Ψ have been made clear for LISP's circumstances. The difficult task is to demonstrate the coherence of defining these two functions independently, especially in what is so widely taken to be a purely procedural formalism. In section 3.a we applied the terms of diagram S3-2 to traditional systems; we next need to examine their applicability to computational calculi in general. It might seem that programming language semantics would provide the formulation of Φ and/or Ψ in the computational case. But this, we will argue, is not so. To show this, we will for a moment set that diagram aside, and will look instead at what traditional programming language semantics is concerned with. This analysis will be undertaken with some care, since the differences between standard denotational semantics and the semantics we will ultimately adopt are crucially important, but nonetheless rather subtle.

3.b.i. Standard Programming Language Semantics

Discussions that defend the utility of formal semantical treatments of programming languages typically cite a number of benefits of this kind of analysis, of which intellectual hygiene is often an underlying theme. It is suggested, for example, that a mathematical account of the semantics of a programming language can provide a rigorous test of how well that language is understood, may enable theorists to prove that implementations of it are correct, can provide a basis on which proofs about the properties of programs may be constructed, and so forth. It is convincingly argued that only a clear semantical formulation can render explicit what the formal structures in a computational system are intended to *mean*, where by "meaning" is implied a general account of the total role that they play as ingredients in a functioning system.

There can be no quarrel with intellectual hygiene, and we do not want to argue with what traditional semantical treatments formalise. In our study of the semantics of LISP, however, we are concerned with a rather different matter: it is our claim that *what LISP is* arises from our attribution of declarative semantics to its structures — that the

programming language, as it has been formalised, represents an attempt to embed in a formal system a variety of intuitions and understandings about symbols and functions already possessed by the typical programmer, that the programmer is expected to attribute to the LISP structures and programs he or she writes. We are attempting, in other words, to make explicit not only *what* computational structures "mean", in the sense of articulating their complete behavioural or computational consequence, but *why* they are intended to mean what they mean. We are trying to get hold of and explicate the understanding that led to the definitions that would be characterised in a semantics of the standard variety. We will not be satisfied, for example, with a crystal clear statement that the atom NIL evaluates to itself in all environments with no side effects: we want to be able to say such things as that NIL evaluates to itself because it is taken to *designate falsity* in all contexts, and because it is accepted as the *standard designator* of that abstract truth value, and because *any* expression that designates a truth value evaluates to the *standard* designator of that true value.

This prior attribution is not explicitly reconstructed in typical semantical accounts, although it permeates those formulated in what is called the *denotational* style. Even there, however, what we will call the designation of symbols is mixed in with a total account of their computational significance, in such a way that what a structure is taken to *designate* is lost in a much larger account of the complete effects a structure may have on the course of an arbitrary computation. All side effects to environment, field, processor, and so forth, are manifested in the single notion of denotation, which is far too broad and inclusive a notion to satisfy our particular requirements. Similarly, the difference, even in an applicative language like LISP, between what is *designated* and what is *returned* is not maintained: the entire analysis is carried out in a mathematical domain where those two entities are typically identified.

In order to make clear how our approach will differ from this tradition, it is well to make some comments on the received understanding of "semantics" in the computational community. As the term is typically used, the *semantics* of an expression refers to the complete computational consequences that the expression will have for arbitrary computations. Thus computer science is by and large behaviourist and solipsistic, in the sense that very little attention is paid to the question of the relationship between symbols and the external world in which the processes are embedded. Thus the main semantical

function is typically of a type that takes complete machine states into complete machine states. This tendency is illustrated by so-called *semantic interpretation rules* for compilers, which deal not with what we take computational structures to designate, but rather with what behaviour they engender (a compilation is clearly judged correct on *operational* grounds, not in terms of semantic attribution).

We may note in passing that this is not the way semantics is construed for natural language. For an English expression the analogous *cognitive significance* of an expression — the complete account of the effects on my head — is by no means the same as the *designation* or *reference* of that term. The two subjects are related: a considerable literature, for example, is devoted to the question of whether cognitive significance will in general have to be accounted for expressly in terms of such designation, or whether it will be possible to account for the internal cognitive consequences without knowing the designation. Thus people argue as to whether an account of the psychological significance of the term "water" will have anything to do with water. However, certainly no one assumes the two subjects can be *identified.* For example, the sentence "Fire!" may have all kinds of consequences on my mental machinery, causing me to abort any other ratiocination I am in the midst of, to send emergency signals to my leg muscles, and so forth. However the word "fire" *designates* nothing whatsoever about my cognitive apparatus: rather, it designates high-intensity oxidation. Furthermore, the fact that it designates fire for me cannot, as many have argued, be explained solely in terms of *behaviour*, certainly not mine, and not of the world either.

In contrast, if one asks of a programmer what the semantics are of some primitive, he or she will typically respond with an account of how such expressions are treated by the primitive language processor. The meaning of QUOTE in LISP is a telling example: the near universal claim as to its "meaning" is that it is a primitive function that defers one level of evaluation. This is quite evidently an account framed in terms of internal computational consequence.

In fairness, there are two subtleties here, which must be brought out. It will turn out, if we analyse programs in terms of their attributed designation (i.e. if we recast computational semantics on our own terms), that many of the *terms* (object designating expressions) of a program will turn out to be designators of elements of the structural field

of another computational process (i.e. they are in some sense meta-structural). In a simple case, for example, the variables and identifiers of, say, a FORTRAN program designate the data structures that form the field of structures over which the FORTRAN program computes. The embedding world of a *program*, in other words, is another set of computational structures — this was the import of the *process model* of computation sketched in section 1.c. From this fact it is easy to see why, if we are not careful, it is apparently consonant with our intuitions to assume that *all* natural semantics remains within the boundaries of computational systems. In addition, most programming languages are typically used in a first-order fashion; thus the explicit designation of terms designating functions can be side-stepped in a semantical account that treats procedure applications as a whole. What remains, typically, are the boolean constants and the numerals, which can be approximately identified with their referents (the truth-values and the numbers); although this last is a little embarassing, it seems the easiest move to make an apparently successful story complete. Equally embarassing are such constructs as closures, which are not quite functions and not quite expressions; they are posited as the "semantics" of procedures, but without a crisp analysis of whether they are *designators* or *designated.*

Once one moves to higher order languages and meta-structural facilities, however, the fundamental contradictions and inadequacies of such an approach emerge. Once one attempts, also, to integrate a representational or descriptive formalism with a procedural one, the same problems come to the surface, for an internal model of semantics for the base level structural field is simply impossible. A purely "internal" semantics, in other words, is simply inadequate as a way of explicating attributed understanding. It is incapable, for example, of explaining that `(+ 2 3)` has to do with addition, or that `λx.x` designates the identity function.

Not all computational semantical accounts are internal, of course; denotational semantics in the Scott-Strachey tradition (as explicated, for example, by Tennent, Gordon, Stoy, and others[3]) deal explicitly with abstract designations — functions and numbers and truth values and so forth. In this respect standard denotational semantics is close in style to the sort of semantics we are in search of. However there is an odd sense in which, for our purposes, it goes too far, making abstract *everything* about the machine, to the point of losing the original intuitions. Consider for example the numbers, which are typically implemented in terms of binary numerals of a certain precision. On *our* account,

expressions like `(+ 2 3)` will *designate* five; the computational consequence of such a term may be that a co-designating numeral is returned. A standard programming language account (except for context relativisation) would *also* map such an expression onto the real number five; thus in this instance they would be allied. Consider however a case of round-off error, or a situation in which integer numerals of only a certain size were supported. We might for example have a `LISP` dialect in which `(+ 18,000,000 19,000,000)` returned the numeral `-27`, rather than `37,000,000`, because of overflow, or where `(= 1.0 (/ 3.0 3.0))` might evaluate to `NIL`, rather than `T`, because of the imprecision of the underlying implementation. In such a circumstance, the kind of semantics we are looking for would make explicit the fact that what was returned did not exactly match what was designated. On a standard denotational programming language account, however, the full designation would be so constituted — if that semantics were precise — to ensure that `(+ 18,000,000 19,000,000)` *designated* the application of a modified kind of addition to the numbers 18 million and 19 million — a modified addition function that yielded the answer -27 on such arguments. Thus the semantics is formulated in service of behaviour, because its goal is to explain behaviour; our goal, in contrast, is to make behaviour subservient to semantics, so that we can decide whether the behaviour is appropriate. Thus we want to know that `(+ 18,000,000 19,000,000)` designates 37 million, so that we can decide whether the numeral `-27` is an appropriate thing to return. If a particular architecture is not constructed so that co-designating numerals are always returned, we are happy to allow that to be *said*, but not at the expense of formulating the pre-computational intuition that enables us to ask whether the result is co-designating or not.

The problem with denotational accounts, in other words, is that they don't identify attribution independent of all the other aspects of an expression's computational significance, and they do not identify that aspect of it that is independent of a procedural or computational processing of the structures. That this is true is evidenced in the fact (and this is perhaps the clearest way to understand our complaint) that there is no way, even when one is given a complete denotational semantics of a language, to ask whether the language processor is designation preserving — no way, in fact, to ask about the semantic character of the processing regimen at all. We too will erect an edifice of the standard denotational variety, but we will not use the word *designation* to refer to the abstract functions that this mathematical structures maps expressions onto. Rather, we will say that

such a theory mathematically manifests the *full computational significance* of a symbol. We reserve the word "designation" because we will formulate an account of that as well; we will then be able to ask how the computational significance accords with the prior attribution of meaning formulated in terms of the independent notion of designation.

> *Denotational semantics, in sum, as currently practiced in computer science is denotational in style, but it is the semantics of what happens to structures, not of what those structures are pre-computationally taken to signify. Because it is essentially operational in character, it does not deal with what programs are about.*

A reader may object that this is too strong a statement: that surely denotational semantics deals *tautologically* with what computational structures *denote*, and that any attempt to discriminate between *designation* and *denotation* is surely splitting hairs. Such an objection, however, would misunderstand our criticism. The point is that "designation" is an English word having to do with what things stand for — a term that arises from the unexplained but unarguable human use of symbols. Denotational semantics would indeed study the proper designation of computational symbols *if it took that designation as ontologically prior to its reconstruction.* In point of fact, however, the accepted technique appears to be this: we are allowed to make the denotation of a computational structure be whatever is required to enable us to characterise mathematically whatever it is we are studying. Consider for example this quote from a recent introductory text:

> *"It will not be satisfactory to take the denotation of a function construct to be the mathematical function defined by its input/output behaviour. To handle side-effects, jumps out of the function's body, etc., we will need more complicated denotations."*[4]

As we have said before, we have no complaint with formulating sufficiently complex mathematical entities to facilitate the behavioural consequences of code that engenders side-effects and jumps. We too will rest on this work, and will use such techniques. However, if our mathematics makes the numeral 5 denote an infinite function from tuples of input/output-streams, processor states, and so forth, onto a function from continuations to outputted answers, we have surely wandered far from any natural notion of what anything stands for. It should take considerable argument to convince any of us that the numeral 5 stands for *anything other than the number that is the successor of four.*

Note in addition, in the quote just presented, the phrase *the mathematical function defined by its input/output behaviour.* Again, this betrays a loss of innocence, this time of a methodological flavour. Surely the input/output behaviour is defined to honour the mapping appropriate *for whatever function the construct signifies.* Surely, that is, if we are within computer science, where we talk of *formal* symbol manipulation. We lose that innocence at the expense of the natural boundaries of the field, admitting car mechanics on an equal footing with *echt* computational practices.

The trouble takes a particular form: the apparent causal relationships — the dependences between theory and practice — are unnaturally inverted. In present practice behaviour rules, and semantics follows. The structure we argue for is the reverse: semantics, we claim, is foundational; behaviour should be designed to honour it. What we understand symbolic structures to signify is primary: we then arrange their procedural treatment to honour this attribution. The input/output behaviour is "what we intended" just in case it honours it correctly; it should be subservient to semantics (as proof theory and derivability and inference rules and so forth are subservient to truth-preservation and entailment and so forth), rather than the other way around.

Although these problems infect our theoretical accounts, lay practicioners have not lost the clarity of semantic innocence. Everyone knows that the numeral 6 stands for the number five; everyone knows that T stands for truth. It is not, in other words, so much that folk pratice is problematic, as that our mathematical meta-theory has lost contact with that native understanding. The reconstruction of lay understanding is thus our task: a goal once again subsumed under our aesthetic of category alignment. Our theoretical account should cohere — should correspond in the boundaries it draws and the patterns it finds — with that of the attributed if tacit understanding that defines the subject matter.

Since the power of our argument will emerge from the increased power of reconstructed systems (not, in spite of these pages, from rhetoric or invective), it is fair to ask what the consequences will be of accepting our view. First, we have denied that standard semantical practice reconstructs what we are calling designation. Note, however, that we have used two words with approximately equivalent meaning: "denotation" and "designation". It is the latter on which we are staking our claims; with the former, therefore, it is only reasonable to be generous. Therefore, in deference to current practice,

we will use the term ***denotational semantics*** to characterise a *style* of mathematical treatment, in which structures are assigned an interpretation in a mathematical meta-language, and in which the formal relationships between such structures are explicitly treated in terms of such interpretations. By denotational semantics, in other words, we refer to a mathematical treatment of the situation pictured in diagram S3-1. What is left unspecified is what *kind* of interpretation is thereby analysed — whether, in other words, the analysed notion of *denotation* has anything to do with the attribution of significance or *designation.*

In our own analysis we will present a variety of denotational accounts, of which two figure predominantly: one of ***declarative import*** (Φ) and one of ***procedural consequence*** (Ψ). It is the first that must, we submit, formulate the *designation* of all expressions. We will argue that denotational semantics of the standard programming language sort is denotational semantics of full procedural consequence mixed with some amount of declarative import, that operational accounts are denotational accounts of procedural consequence, that Tarski-like model theories for logical languages (such as for the first-order predicate calculus)[5] are denotational semantics of declarative import, and that a proper reconstruction of LISP requires both such treatments. In order to demonstrate that we have satisfied the third mandate listed at the beginning of this chapter, in particular, we will have to have both semantical treatments explicitly available.

It may seem odd to the reader, especially one familiar with the logical traditions, to call the relationship Ψ a *semantical* one. More particularly, it might appear that what we are calling *declarative semantics* is merely what in logic is called *model theory*, and what we are calling *procedural semantics* is what in that tradition is called *proof theory.* Model theory, after all, deals with the declarative interpretation function and with satisfaction and designation and all the rest; proof theory deals with the relationship between sentences provided by the inference processes. However this comparison is too facile, and fails to recognise a crucial point. Perhaps in part because derivability is not a *function*, there is no tendency to treat the procedural relationship in logic in terms of *attributed* understanding: rather, one formulates and understands it purely in terms of the mechanisms that implement it. The entailment relationship is in contrast semantically characterised, but it is so simple, easily stated, and so purely a corollary of the main declarative semantical treatments — i.e. of the model theory — that it is not given a name on its own. In our

computational formalisms, however, we do understand procedural consequence in terms of attributed understanding: as we made clear at the outset, *we understand LISP in terms of function application*, and function application is an essentially non-computational, and therefore attributed, understanding, deserving of its own semantical analysis. There is an entirely non-attributed understanding of how the procedural treatment *works*, and the correspondence between the two constitutes the proofs of soundness and completeness and the rest for the relationship Ψ.

An example will make this clear. Consider the expression `(* (+ 4 2) (- 4 2))`. The declarative semantics will tell us that this structure designates the number twelve. The procedural *semantic* characterisation in, say, a depth-first left-to-right designation-preserving computational system like 2-LISP, would say that this generates the application of the numeral-addition function to the the numerals 4 and 2, followed by the application of the numeral-subtraction function to the same two numerals, followed by the numeral-multiplication of the resulting numerals, yielding in the end the numeral 12. This last is not the designation function (even though analogous function applications are used in the meta-theory to specify the designation — an entirely different affair), since it talks of operations and results and temporal ordering and so forth. Neither, however, is it the *formal symbol manipulation* account that is the true computational story of what happens, which has to do, as we have said so often, with structure and environments and processor states and intermediate results, none of which makes reference to the notions of functions and application at all. Rather, it is a semantical account, probably compositional and so forth, *of what happens.*

It is Φ, in other words, that would map `(* (+ 4 2) (- 4 2))` onto the number twelve; it is Ψ that would characterise the relationship between `(* (+ 4 2) (- 4 2))` and the resultant co-designating numeral 12, in terms of normal-form codesignators and function application. Finally, it is the computational account *of the implementation* that would specify in fact what happens when `(* (+ 4 2) (- 4 2))` is processed: an account, presumably, provably equivalent to that semantically specified in the formulation of Ψ. These three related but independent stories — of *designation*, of *procedural consequence*, and of *implementation* — will permeate the discussions throughout the entire dissertation.

It will turn out, as the reader will see in the first parts of chapter 4, that making constant reference to both of Φ and Ψ quickly becomes cumbersome in dealing with a real system — even one as limited as the pure LISP dialects we will develop. More importantly, it becomes *unnecessary* if Φ and Ψ are sufficiently closely allied that talk of one can always be simply converted to talk of the other — it is made unnecessary, in other words, exactly when one succeeds in developing a semantically rationalised system. For example, in 2-LISP, because of the semantical type theorem, and because of the category alignment between Ψ and Φ, it is always natural to talk only of the designation (Φ) of formal expressions; their procedural import Ψ is so readily obtainable from their declarative import that intricate discussions of the former are happily dispensed with. From this fact, however, it should not be concluded that Ψ is irrelevant: the very point is to make Ψ — a function that one necessarily must contend with — so consonant with Φ that it can be safely ignored. Our ability to ignore Ψ in most of our thinking about 2-LISP, in other words, will be our great success, just as the ability to prove that ⊢ and ⊨ are equivalent in complete logical systems allows one to think in terms of just one. In dialects in which procedural treatment does not parallel the declarative treatment in systematic ways, the luxury of using just one cannot be achieved.

3.b.ii. Declarative Semantics in LISP

It might seem to take some care to show that programmers bring a pre-computational attribution of significance to LISP, but in fact it is straightforward, once it is clear what the endeavour is. Some simple LISP examples will illustrate. The LISP atom T, for example, is taken to signify truth, and the numeral 6 to signify the number five. Similarly, the expression (CAR X) signifies the first element of whatever list X signifies. These claims do not rest on the fact that the atom T evaluates to itself, or that the expression (EQUAL 'A 'A) evaluates to that atom; rather, the situation is just the reverse. We make the atom T evaluate to itself, and (EQUAL 'A 'A) evaluate to T, *because T stands for truth.* Similarly, the numeral 5 does not signify the number five because of how it is treated by the LISP "addition" function. Nothing but confusion would result if the expression (+ 5 6) were treated by the LISP interpreter in a way that bore no relationship with our understanding of that expression as a term designating the sum of five and six. There is nothing saying that LISP *has* to be defined this way, but the fact remains that it *is*

designed in this way, and for good reason. Even though the LISP interpreter does not know that the numeral 5 designates five, it is enormously useful that we define its behaviour so that we can make use of our externally attributed understanding that 5 stands for five. We live happily with the fact that LISP deals with numerals (and not with numbers) because we can satisfy ourselves that things have been arranged so that no differences in behaviour arise. But to be able to say "it is just the same either way" implies that we know the difference, and that we understand one as standing for the other.

Imagine instructing a novice in the use of LISP — a useful gedanken experiment because it provides a natural setting in which one's pre-theoretic intuitions need articulation. One would clearly mention the fact, if the student did not realise it straight away, that the atom T stood for "true", and NIL for "false", before attempting to explain why the expression (EQUAL 3 4) evaluates to NIL. Similarly, one might say that the LISP field of data structures included linked structures called "cons-cells", and that the first half of such a cell is called its "CAR"; the second half, its "CDR". By using such *terminology* in English — the paradigmatically *referential* language (and not, at least so far as anyone has shown, a *computational* language) — one legitimates the use of such descriptive phrases as "the *CDR* of CELL-20", and so forth. Thus we might say to him or her, "*If the CAR of this list is the atom LAMBDA, then we know that the list represents a function ...* ". This is an entirely natural way to speak, which again betrays the fact that in *our* use of the terms "CAR" and "CDR", we think of them as *concepts under which to form descriptions*, not as the name of procedures. And, at the risk of being repetitious, we need to remember that *descriptions* and *functions* are different categories of object. The phrase "*the largest integer N such that N is its own square*" is a description, but invokes no procedure.

A striking piece of evidence that we understand (CAR X) to signify the first element of a list, prior to our understanding that the LISP interpreter will *return* the first element of that list when it *evaluates* the expression, is provided by the SETF procedure (recently introduced in MACLISP[6]). A generalised assignment operator is defined such that the expression (SETF (CAR X) <EXP>) is equivalent to (RPLACA X <EXP>) (similarly (SETF X 4) is equivalent to (SETQ X 4), (SETF (CADR Y) Z) is equivalent to (RPLACA (CDR Y) Z), and so forth). SETF doesn't evaluate its arguments — rather, it is a complex macro that *unwinds* its first argument, so to speak, constructing a modified compositional structure that will effect the change on the proper structure.

The code for SETF could be used as a way of explaining what SETF means, but this doesn't answer the question of how SETF is understood, or why it was defined, or what was in the mind of the person who defined it. One sometimes hears the explanation that SETF is a procedure such that after evaluating (SETF A B) then evaluating A will return B, but this is far too non-specific to capture its meaning. By this account (SETF (CAR X) 4) could expand into either of (DEFINE CAR (LAMBDA (X) 4)) or (SETQ X '(4 5 6)). In response to such criticisms, partisans sometimes offer the reply that SETF effect the *minimal* change necessary to make the first argument evaluate to the second, but of course the notion of minimality would have to be discharged, and is probably inadequate no matter how it is construed. In sum, all of this kind of talk is an inadequate reconstruction of the intuitive feeling that SETF should change the *structure that the first argument points to, in some sense other than what it evaluates to.*

Another way in which such constructs are sometimes explained is in terms of how they work. One sometimes hears of *left-hand side* values and *right-hand side* values, since the the non-evaluative situation typically occurs on the left hand side of the grammatical expression used for such assignment. Such a characterisation, of course, is inelegant in the extreme. A better account, but one still tied unnecessarily and unhelpfully to the mechanics of implementation, uses a notion of a *locative*: thus X in the expression (SETF X '(4 5 6)) would be used as a locative to identify a *location* to be set to the quoted list. This too, however, is an admission of defeat: the name of a mechanism used to implement a simple intuition is used as the theoretical vocabulary in terms of which it is defined, for lack of a better alternative. We did not need any notion of *location* in defining LISP in the previous chapter; it would be odd to introduce one at such a point. Furthermore, the concept of locations would seem to arise from Von Neuman architectures, and LISP is powerful for, among other reasons, the fact that its abstract virtual machine is in no way dependent on notions derivative from this class of computational device. Furthermore, the actual code that implements SETF does not make reference to the fact that LISP is *implemented* in terms of locations on such a Von Neuman machine; it would be odd, therefore, to think that the natural *explanation* of SETF would need to depend on this inaccessible underlying architecture.

There is a much simpler explanation of SETF than its code, that again betrays the fact that we use our understanding of language to understand formal systems. SETF works in

the way that it does *because it treats its first argument as an intensional description*, in what Donellan has called an *attributed* sense.[7] It is just like the use of the phrase "the President" in one reading of the sentence "*Lower the President's salary to $30,000*", where we mean to decrease the compensation of *whoever holds the office*, not of the person who is currently President independent of occupation. The phrase "the President", in this construction, is not used purely extensionally; if Mr. Glasscock were President when the phrase was uttered, it would not (at least on the reading we are considering) mean that we specifically meant to lower that fellow's salary. Rather, we mean to refer to something like *whoever satisfies the description "the President"*. Similarly, (CAR X) is not used in a purely denotational sense in (SETF (CAR X) 3); we mean something like, given some value of X, make the minimal change such that the *intensional description (CAR X)* will designate the numeral 3.

If (CAR X) was meaningful only in terms of its behaviour under EVAL, this would be a difficult protocol to defend or explain. But it is easily comprehensible to a human, because of the fact that we understand (CAR X) to be a composite *term* — a description of the first element of the list X. We don't, of course, know what it is to use a description intensionally: the answer awaits the millenial epistemologist. But it is undeniable that we *do* use language this way, and it therefore becomes perfectly natural to invent computational constructs (like SETF) that use other computational descriptions in an analogous fashion. But to accept this means we accept the fact that (CAR X) is a *designative term*, not simply a procedurally defined form that returns the first element of a list. *By accepting SETF, in other words, we are admitting the pre-computational* (and language derived) *attribution of meaning to computational structures.*

This dissertation constantly skirts the crucial — but yet to be understood — issue of intensionality, which permeates this example. The term (CAR X) is being used intensionally in the SETF context. There are other such examples throughout computation. The construct, for example, whereby one variable is hooked in some manner to another (such as assigning Y to "always" be X + 1, where that is intended to mean that Y should be constrained to be one greater than X, no matter what X subsequently becomes — i.e. that Y should track X, remaining exactly 1 greater than it), similarly uses computational structures as descriptions, in intensional contexts. Similarly, the recently emergent *constraint* languages[8] are rife with designative expressions. All of these are practices lifted from the

lay use of language. As we understand how to embed them coherently into computation systems, we do so, thereby making our programming languages more like natural languages, and therefore making computational systems easier to use. Our present claim is merely that this practice should be admitted, and then to use the best understanding — the best theories and conceptual analyses — of linguistic and epistemological phenomena in understanding that computational practice.

The moral of these few examples is that we have an understanding of what LISP expressions signify that is *prior to our understanding of how they are evaluated*, and furthermore, that the study of *human language* will play a role in uncovering that prior ascription. The evaluation process is elegant to the extent that it does something that coheres with that prior understanding — and as we will see in this chapter, 1-LISP's does a reasonable but not excellent job, failing particularly in meta-structural circumstances. Our primary task, therefore, is, so far as it is possible, to make explicit that prior understanding, without making reference to EVAL or to how the interpreter works in establishing this semantical attribution. We cannot, in other words, answer a student's question of the form *"What does the expression (LAMBDA (X) (+ X Y)) mean, given the occurrence of the free variable Y?"* with the response *"Well, if we look at the definition of EVAL we can see that it notices that the first element is the atom LAMBDA and constructs a closure"*. Rather, the point is that we have to establish the semantical import *separately*, in order then to be able to characterise the evaluation process in terms of it. Unless we can do this we will have no principled reply to our student's next question: *"Why does EVAL work in this way?"*.

Sometimes this search for a purely declarative reading of LISP exprssions will fail. It is difficult to say, for example, what, if anything, the construct (GO LOOP) or (THROW 'EXIT NIL) or (QUIT) *designates*. Nonetheless, we will attempt to do as thorough a declarative treatment as seems part of our natural understanding. Even in the three expressions just given, for example, the *arguments* are clearly first and foremost designators, rather than structures with natural procedural consequence. The more important lesson is that, to the maximum extent possible, a calculus in its very design should facilitate such declarative attribution, since it is apparently part of our natural way of comprehending formal systems, even those that are computationally oriented.

(In order to avoid confusion, we should remark here that the foregoing argument does not imply that whereas statically scoped free variables will succumb to a declarative treatment, dynamically scoped variables will not. Admittedly, a pre-procedural treatment of designation is possible for the λ-calculus, and this is why the λ-calculus is lexically scoped — it is the only obvious protocol in a formalism with declaratively specified function designators. Nonetheless, declarative import and statically (pre-computationally) specifiable semantics are independent notions, as discussed in more detail in section 3.c.v.)

A final comment. There can be no argument that our focus on an applicative calculus — and on *designational* attribution — betrays the fact that we are allying computational constructs with pre-computational notions of *noun phrases.* We are, in particular, taking expressions to be *terms,* and functions are playing the kind of role that descriptive concepts do in English. Thus we have little to say of interest about side-effect operators in LISP, like SET and so forth — constructs unarguably closer in intended interpretation to *verb phrases* in natural language. Our basic moral is that computational concepts should be related to natural language constructs, since, on our view, it is in their tacit correlation with natural language that much of their coherence lies. It is clear, however, that this correlation includes natural language formations of a wider diversity than simple designating nominal phrases. The obvious extension of the approach we have followed here, therefore, would extend the style of analysis that we have given to nominal designation, to include other aspects of the natural structure of human language. This author has long felt that a Gricean[9] speech act analysis of ALGOL would uncover much of the tacit structure of computational practice; the present investigation can be viewed as a tentative step towards such a full reconstruction.

3.b.iii. Summary

In LISP's case, the function computed by EVAL is clearly Ψ; the semantical function formalised by standard mathematical semantics is a mixture of Φ and Ω; the pure designation function Φ is not normally formulated. Our strategy will first be to articulate a defensible account of 1-LISP's Φ, then to explicate 1-LISP's Ψ with reference to the operational style of account given in the previous chapter, and then finally to inquire as to what semantically-definable function Ω the procedural function Ψ might be the correct embodiment of. It will be at this point in the analysis that the form of the evalutaion

theorem will be articulated and shown to hold of traditional LISPs. It is the inelegance of 1-LISP's Ω that will lead to the suggested design for a clean *prior* definition of an appropriate Ω, and then a rationalised Ψ that provably implements it, satisfying the normalisation theorem.

The following table, by way of review, summarises the characterisations we have made about a variety of systems. Note in particuar that 1-LISP and SCHEME (and by implication all extant LISP dialects) are without well-specified versions of Φ and Ω; 2- and 3-LISP are not so much *new* as they are *traditional*, in postulating interpretation functions of the classic sort.

A Semantical Characterisation of a Variety of Formal Systems (S3-12)

System	Φ	Ψ	Ω
λ-calculus:	Declarative interpretation function	α- and β- reduction	Normal-form (preserves Φ, no further reductions apply)
Logic:	Declarative interpretation function	$\vdash$	$\models$
1-LISP, SCHEME:	?	Evaluation	?
2-, 3-LISP:	Declarative interpretation function	Normalisation	Normal-form(preserves Φ, form determined by semantic type)

3.c Preparations for 1-LISP Semantics

There are still a few preparations to be made before we can sketch appropriate declarative and procedural semantics for LISP. Some of these have to do with general issues; some with the relationship between Φ and Ψ; some with LISP's particular circumstances. Though it is unfortunate to spend an entire section constructing machinery, it will make the subsequent technical manoeuvring much more straightforward.

3.c.i Local and Full Procedural Consequence

We have presented the function Ψ as if it were the function that made manifest the full procedural consequence of each symbolic structure, but that is an over-simplification. Ψ is the function computed by the language processor that takes each expression or formal structure into the expression *returned* as its "value" or whatever — this is at the crux of LISP being an *applicative* language. In point of terminology, we will call $\Psi(x)$ the ***result*** of x, and will say that x ***returns*** $\Psi(x)$, and that Ψ defines the ***local (procedural) consequence*** of an expression. During the course of the processing, however, there may in addition be what are known as *side-effects* on the state of the machine. Two kinds of side effect, in particular, need to be handled, as intimated in chapter 1: alterations to the structural field *F*, and alterations to the processor, as expressed in the environment *E* and the continuation *C*. A mathematical treatment of the full procedural consequence of an expression, therefore, will have to reflect not only what result is returned when the expression is processed, but also any effects it may have had on these other components of the abstract machine.

In a standard semantics of a programming language, such effects are dealt with by making the main semantical function be a function not only of the formal structure, but also of "the rest of the machine": usually a memory, an environment, and a continuation (input/output effects, in addition, can be treated by including some appropriate abstract object — such as streams — but we will ignore peripheral behaviour at present). Given our reconstruction of computational processes as consisting of a structural field and a processor, and our claim that the two theoretical entities of an environment and a continuation adequately characterise the state of a processor, we can see how this standard

treatment effectively makes the main semantical function take complete machine states into complete machine states. In other words, if our field and processor model of computation is adequate, such a semantical function will *necessarily* be sufficiently powerful to allow arbitrary computational consequences to emerge from the processing of expressions or structures.

We too will have to formulate such a complete state-transforming function in order to characterise the ***full procedural consequence*** of 1-LISP processing. We will call such a function Γ; our initial version will be of type ***[[S × ENVS × FIELDS × CONTS] → [S × ENVS × FIELDS]]***. Certain types of expressions, such as non-local control operators (THROW and CATCH and FRETURN), structure modifiers (RPLACA and RPLACD), and so forth, will be comprehensible only in terms of their full Γ-characterisation. However we will try to focus primarily on the simpler function Ψ in analysing 1-LISP evaluation, since it is with respect to this simpler function that our criticisms of 1-LISP will be formulated. We have no complaint, in other words, with the fact that the processing of the 1-LISP expression (RPLACA '(A B) 'C) alters the structural field in such a way that CAR of the first argument is changed from the atom A to the atom C, and it is unarguable that this fact can only be made explicit when looking at Γ(RPLACA). Our only comment is that it is important to retain the function Ψ as a valid subject of study, since it is the coherence of Ψ with Φ that we wish to scrutinise. In addition, we will attempt to formulate Γ in such a way that the function Ψ will play a self-evident role as an ingredient.

The complete state-to-state transformation function, in other words, yields for our purposes too *coarse* an analysis; our complaints with LISP, and our models for reconstruction, emerge only from a finer grained decomposition of a computational symbol's full significance. In addition, as of course may reasonably be expected, it will turn out that our attempts to define Φ and Ψ *independently* of Γ will founder over the questions of side-effects; in our second pass (section 3.e) we will define a more complex function Σ — a variant on Γ — with Φ and Ψ integrated more fully into it. On our initial attempt, however, in the general case, Γ will be given an expression, an environment, a field, and a continuation (continuations are of type ***[[S × ENVS × FIELDS] → [S × ENVS × FIELDS]]***):

```
∀S ∈ S, ∀F ∈ FIELDS, ∀E ∈ ENVS, C ∈ CONTS                    (S3-13)
    [∃S' ∈ S, E' ∈ ENVS, F' ∈ FIELDS
        [Γ(S, F, E, C) = <S', E', F'>]]
```

Given this general characterisation, we can make precise some of the ingredient concepts we will ultimately use in defining an adequate notion of *normal-form.* In particular, an expression s will be called ***side-effect free*** just in case its Γ-characterisation is as follows (we often write ΨEF(<exp>) and ΦEF(<exp>) for ((ΨE)F)(<exp>) and ((ΨE)F)(<exp>), respectively):

```
∀F ∈ FIELDS, E ∈ ENVS, C ∈ CONTS                                    (S3-14)
    [Γ(S, F, E, C) = C(ΨEF(S), E, F)]
```

There are two ways in which an expression s can fail to be side-effect free: it can affect the *field* and it can affect the *processor.* In the first case we say that an expression has a ***field side-effect***; its full procedural consequence would have the following structure:

```
∀F ∈ FIELDS, E ∈ ENVS, C ∈ CONTS                                    (S3-15)
    [Γ(S, F, E, C) = C(ΨEF(S), E, F') for some F' ≠ F]
```

Processor side-effects are of two types: those that affect the environment, and those that do not invoke the regular continuation. More precisely, if an expression has a ***environment side-effect*** then the environment yielded up as a result of processing will be different from that in which it was processed:

```
∀F ∈ FIELDS, E ∈ ENVS, C ∈ CONTS                                    (S3-16)
    [Γ(S, F, E, C) = C(ΨEF(S), E', F) for some E' ≠ E]
```

Similarly, if an expression s has a ***control side-effect***, then the continuation c would not be the function given the result. Informally, we would characterise this as follows:

```
∀F ∈ FIELDS, E ∈ ENVS, C ∈ CONTS                                    (S3-17)
    [Γ(S, F, E, C) = C'(ΨEF(S), E, F) for some C' ≠ C]
```

This does not, however, capture the intuition; it is too liberal, since too many functions c' can be found for side-effect free expressions. We need instead the following:

```
∀F ∈ FIELDS, E ∈ ENVS                                               (S3-18)
  ¬[∃F' ∈ FIELDS, E' ∈ ENVS
      [∀C ∈ CONTS [Γ(S, F, E, C) = C(ΨEF(S), E', F')]]]
```

At various points we will hint at the appropriate mathematical characterisation of the full procedural consequence of side-effect expressions (non side-effect ones need be characterised only in terms of Ψ since S3-14 supplies the remaining information). Our main interest in Γ, however, in so far as we are able to avoid complexities, will be in showing that all normal-form designators are side-effect free.

It will turn out, however, that the ultimate definition of Φ will have to refer to Γ, because of interactions between the declarative import of composite expressions and the potential side effects engendered by the procedural treatment of their consituents. The following expression, for example, will have to be admitted as designating the number five, rather than the number three:

```
(LET ((A 1))                                                    (S3-19)
    (+ 2 (PROG (SETQ A 3) A)))
```

The only way in which this can be made explicit, it is clear, is in terms of Γ. For the time being, however, we will ignore these potential interactions; they will surface in section 3.e.

3.c.ii. Declarative Semantics for Data Structures

In most programming languages, the set of expressions that are *programs* and the set that are *data structures* are kept distinct. LISP, of course, is distinguished partly because it does not make this distinction, and consequently gives a program the ability to manipulate program structures directly. Such a capability is clearly a prerequisite to the construction of a reflective dialect — it is not incidental, in other words, that we are studying a language with such a property. What this does, however, is to raise a question as to whether a subsidiary distinction between program and data structure can be raised *in particular.* Can we, in other words, distinguish a particular structural field fragment that is intended to be a "program" from one intended to be "data"?

A considerable literature documents the fact that making such a distinction is problematic at best, and possibly ill-founded.[10] No worthy account of the distinction between the two concepts *program* and *data*, if indeed they are coherent and exclusive, has been formulated. It is of some interest, therefore, to note that the present endeavour of making explicit the attributed semantics to all computational structures may in fact serve to reconstruct this notion as well. In particular, it is tempting to *define* as a ***program*** any section of structural field with two properties: procedural consequence Ψ is defined over the fragment, and the declarative import Φ of all terms within the fragment maps terms onto other structural field elements. We have in other words the following claim:

All terms in programs are meta-structural designators.

Such speculation, however, — as to whether such a definition adequately captures the persistent lay notion — is beyond the scope of this thesis. For present purposes, we will not make any such distinction.

Although this undifferentiated position is in some ways simple, a problem thereby arises in our formulation of semantical interpretation functions, as to whether they should apply to *all* structural field elements, or merely to those "considered to be programs" or "considered to be data structures". It seems particularly troublesome regarding the procedural functions Ψ. The simplest approach is to posit a function defined over *all* structures, but to use it in our analysis only over such structures as are intended to have procedural consequence, and to ignore whether or not it is defined over data structures, or, if it happens to be defined, to ignore what it maps them into. This is in fact the tack we will adopt, regarding Ψ.

The situation with respect to Φ is more complex. While it is clear that straightforward, paradigmatic data structures do not bear procedural consequence, there is a question as to whether we want the theory of declarative import we use to explain terms in program to hold of all data structures as well. In other words, if we say that the structure `(CAR A)` designates the value of the function designated by the atom `CAR` applied to the referent of the term `A`, are we committed to saying of any data structure `(F G)` that it designates the value of the function designated by `F` applied to the referent of `G`. Suppose, for example, that a user of the language sets up a list of students and grade point averages, of the following format: the whole list consists of a list of two-element lists, of which the first element is the student's social security number, and the second element is grade. Thus we might have `((234-23-2344 3.95) (021-99-8276 4.0) ... )`. It would seem, if we adopt the view that our declarative interpretation function applies uniformly to all structures, that it would claim that, semantically, each element of this list designated the value of the function designated by the social security number applied to the grade point average, which is of course nonsense.

However there are two reasons this need not bother us as much as it might seem. First is the standard confusion between lists being used to designate sequences or enumerations, and lists used to encode function applications. Once we have made this into a structural distinction, the foregoing example would be expressed using the enumerative

type of data structure; in 2-LISP it would be expressed as the structure [[234-23-2344 3.96] [021-99-8276 4.0] ...]. The standard semantics (for 2-LISP) will say of each doublet in this structure that it designates the abstract sequence consisting of the referent of the symbol encoding the social security number and the referent of the numeral encoding the grade point average, which is just what one would like.

The second reason suggesting that it is indeed appropriate to posit the declarative semantics over all data structures comes from our general endeavour to unify procedural and representational systems into a coherent single framework. As discussed in section 3.c.v, below, subjecting data structures to a declarative semantics is still far removed from making the data structures an adequate declarative language, but it is nonetheless a valid first step in that direction. It is much clearer in the case of data structures than in the case of programs that we always understand them by attributing declarative import to them: it is inconceivable that there could be a useful data structure for which one could claim to understand it, but could say nothing about what it stood for or represented.

There are some consequences of this approach deserving notice. First, we are implicitly admitting that the class of structures falling in the natural domain of Φ is wider than that falling under Ψ. Secondly, whereas the range of the function Ψ is the set of structural field elements S (since Ψ is of type $[S \rightarrow S]$), the range of the declarative intepretation function Φ will be much larger (since Φ is of type $[S \rightarrow D]$). Already we have assumed that the semantical domain D includes numbers, sets, and functions, as well as all the elements of S; this last move of including all user data structures under its purview means that we will have to allow it to include the user's world of objects and relationships. This, however, poses no particular problem.

3.c.iii. Recursive Compositionality, Extensionality, and Accessibility

The third comment has to do with what is known as *recursive compositionality.* Typically, in formulating the semantics of a base level language, the model theorist has no idea and no interest in what *particular* predicates and objects the user of the language will employ. The mathematical semantics merely assumes that each predicate letter is taken to signify *some* predicate, and each constant to signify *some* object in the semantical domain. The task of the model theorist is generally taken to be one of showing how the significance of *composite* expressions arises from the significance of those expressions' *constituents.*

There is no guarantee for all meaningful languages, of course, that the meaning of the whole is in any systematic way determinable from the meanings of the parts, but there is certainly some sense in which this kind of compositionality at least roughly holds in natural language, and it is made to be true in all formal languages.

In addition, recursive compositionality semantics of this sort is a powerful way of formalising meaning, and there are various devices (passing environments and continuations explicitly in programming language semantics is a good example) by which a great deal of context-relativity of the meaning of an expression can be captured within the basic compositional framework. We too will adopt a recursively compositional stance, in formulating all of Φ, Ψ, and Γ, in the traditional fashion. The bulk of the emphasis, particularly in this chapter, as we attempt to formulate the *style* of semantics we want to adopt in the remainder of the dissertation, will be on the various different semantical relationships. However the reader can expect that, unless otherwise noted, Φ of a composite expression will be defined in terms of Φ of its constitutents, and similarly with Ψ and Γ.

Recursive compositionality should not be confused with what is known as an *extensional* semantics. If X is a composite expression comprising three ingreedient expressions A, B, and C, then if Φ(X) is a function only of Φ(A), Φ(B), and Φ(C), then each of A, B, and C are said to occur within X in an *extensional* context. It follows that if some further expression D had the same Φ-semantics as A, then an expression X' formed from X by replacing A with D would have the same interpretation under Φ. For example, (+ 2 3) and (+ 2 (- 4 1)) signfiy the same number 5, since arguments to the addition function are extensional, and (- 4 1) signifies the same number as does the numeral 3.

It is different to say of a composite expression X, however, that its interpretation under some semantical function is a function of its ingredients, since that means, to use the previous example, that Φ(X) could be a function not only of Φ(A), Φ(B), and Φ(C), but also of A, B, and C themselves. We will say that in this case the semantics of X are still recursively compositional, but that X is not *extensional* (more precisely: that the constituents A, B, and C do not occur in *extensional contexts*). For example, the single argument position in the LISP expression (QUOTE X) is not extensional; nonetheless, the semantics of (QUOTE X) is still compositional. In order for X to fail to be recursively compositional Φ(X) would have to depend on some quite other factors, such as on the time of day or on X's

position in a data base — facts that are not part of its own constitution.

In lexical systems, where these notions have been developed, the notion of an "ingredient" is clear — given for example by the derivation tree of the lexical grammar. However it is not immediately obvious that the notion of *ingredient* or *constituent* is in general defined over arbitrary structural field fragments (and, as we have several times pointed out, it is the structural field, rather than the notation used to signify it, that is the source domain of both Ψ and Φ). Without it both concepts of compositionality and of extensional contexts are ill-formed. Thus, although we will want to say of LISP that its semantics are compositional, we need to show that such a claim is meaningful.

It happens that in the LISP case we have a relatively straightforward answer to this issue. In the previous chapter we discussed the accessibility relationship on the field: we can say of expression S that its *constituents* are those structural field elements accessible from it (except for the property lists accessible from atoms):

$$\text{CONSTITUENT} : [[S \times S] \rightarrow \{\text{Truth, Falsity}\}] \qquad \text{(S3-20)}$$
$$\equiv \lambda S_1, S_2 \in S[[S_1 \in \text{ACCESSIBLE}(S_2)] \wedge [S_1 \neq \text{PROP}(S_2)]]$$

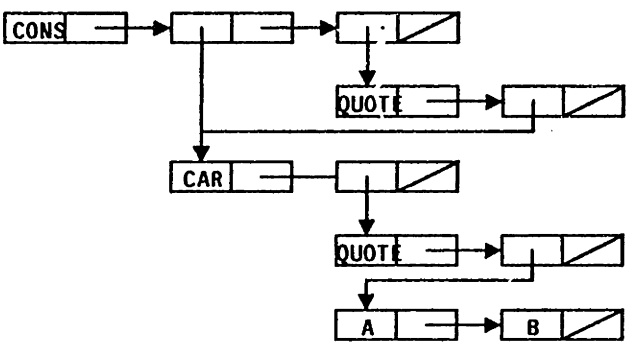
Such an accessibility relationship, however, does not include the accessibility derived from the environment: thus in a recursive definition the binding of the recursive name within the body does not yield a circular constituent structure. It is possible, however, for a structure to be accessible from itself — many examples were given in the last chapter. The recursive definition of a multiplier given in S2-111, for example, is by this definition one of its own constituents. Thus the notion of *compositional* semantics is at best partially defined in the LISP case.

Note as well that we cannot ask of a particular token structural field fragment whether it occurs in an extensional context or an intensional context, as if a single answer were always forthcoming. A given expression may occur in more than one context, since it may be accessible from more than one other structure. Even in as simple a structure as (CONS X (QUOTE X)) (the expansion of (CONS X 'X)), for example, there is only a single atom X; one cannot ask whether X occurs extensionally or intensionally. Nor is this restricted to atoms; in the structure which would be printed as:

```
(CONS (CAR '(A B)) '(CAR '(A B)))                    (S3-21)
```

where the two lists beginning with CAR are the same — i.e. where there are shared tails, as diagrammed in S3-22 below — the very same token s-expression is both extensionally and intensionally used.

(S3-22)



3.c.iv. Structure vs. Notation

The fourth comment has to do with the source domains of Φ and Ψ. Φ, being a semantical function, maps formal structures onto a signified world. The source domain in this case will be the structural field *S*, not the notational domain *L*, as is more commonly the case in logic and traditional programming language semantics. Furthermore, this distinction is not simply one of treating the notational structures *abstractly* (i.e., as lexical *types*), rather than as concrete lexical items: *S* is not merely the abstract syntax of *L*. The elements of *S*, as we have already seen in detail for 1-LISP, are not even type-identical with derivation trees for the grammar of the lexical notation. As we suggested in that chapter, and as was pictured in S3-2, there is an entire independent semantical account Θ relating notational expressions to elements of the structural field. Similarly, Ψ maps elements of *S* into elements of *S*, not elements of *L* into elements of *L*.

There is a minor difficulty arising from the fact that it is the structural field over which our semantical functions should range, having to do with the form of our meta-language. It is traditional to have the meta-language include the object language, or at least to enable meta-linguistic expressions to contain quoted fragments of the object language. It is straightforward to say λS.YS where S is to range over s-expressions, but we cannot *quote* s-expressions in lambda-calculus notation, *since s-expressions are not notational objects.* The temptation is to use 1-LISP *lexical notation*, as for example in λF.F("(CONS 'A 'B)"), where the quoted fragment is 1-LISP notation. However if we were to proceed in this way we

would have to embed the entire theory of notational semantics Θ_L within the accounts of Ψ and Φ, which would complicate matters tremendously.

Our solution — albeit a partial one — will be to use a single occurence of a double quote mark (by analogy with LISP's own single occurence of a single quote mark to notate LISP internal quotation) in the meta-language, followed by *italicised* 1-LISP notation, as a structural device intended to express a designator *of the 1-LISP s-expressions for which that notation is the lexical notation.* The meta-linguistic phrase "*X*, in other words, will be taken as a meta-linguistic abbreviation for Θ_L('X'). We will note where this convention is insufficient, such as in cases where the lexical notation is ambiguous or incomplete.

In addition, we will use Quinean "corner-quotes"[11] to quote those expressions *in the meta-language* with schematic variables; occurrences of the variables in question within the scope of the quasi-quotation will be underlined. Thus for example, the expression

$$\forall X\ [[\ X \in \{A,\ B\}\ ] \supset [\ F(\ulcorner G(\underline{X})\urcorner)\ ]] \qquad \text{(S3-23)}$$

is extensionally equivalent to

$$[\ F('G(A)')\ ] \wedge [\ F('G(B)')\ ] \qquad \text{(S3-24)}$$

In addition, we will use a combination of the corner quotes and the double quote mark convention just established in an obvious way. Thus

$$\forall X\ [[\ X \in \{"A,\ "B\}\ ] \supset [\ F(\ulcorner"(G\ \underline{X})\urcorner)\ ]] \qquad \text{(S3-26)}$$

is extensionally equivalent to

$$[\ F("(G\ A))\ ] \wedge [\ F("(G\ B))\ ] \qquad \text{(S3-26)}$$

It follows that the previous convention would more properly be stated as follows: meta-linguistic expressions of the form $\ulcorner"\underline{X}\urcorner$ will be taken as abbreviatory for expressions of the form $\ulcorner\Theta_L('\underline{X}')\urcorner$. The similarity between this meta-linguistic protocol and the backquote mechanisms in the LISPs we consider is striking: in both cases a quasi-quotational style is used, with those elements that are terms from the meta-language, not from the quoted expression, especially marked.

There are structural field elements for which no lexical notation exists; it follows that the protocol just adopted is not fully general. Although in our brief sketches in the present document we will not require systematic meta-linguistic reference to non-notatable

structures, a more cumbersome but fully general device is always available, using explicit CARs and CDRs. Since those functions are encoded in the meta-language as the first and second coordinates of fields (neither CAR nor CDR is a valid meta-linguistic function), [P(*(G A B))] could equivalently be expressed, given an appropriate F, as:

$$\exists S\,[[P(S)] \wedge [F^1(S) = {}^{\#}G] \wedge [F^1(F^2(S)) = {}^{\#}A] \wedge [F^1(F^2(F^2(S))) = {}^{\#}B] \wedge [F^2(F^2(F^2(S))) = {}^{\#}NIL]] \qquad \text{(S3-27)}$$

This is also relevant if there are side effects to the code itself. If, for example, X is bound in E_1 to (RPLACD X '(Y (3))) and Y to (1 2), then after processing X, X will be (RPLACD Y (3)), but Y will still be (1 2), not (1 3). After a *second* processing of X, Y would be changed to the latter value. All of this will fall out of the semantics; to illustrate such an example, however, we would have to use the notational style of S3-27, rather than using the pseudo-quotation operator just introduced.

3.c.v. Context Relativity

A fundamental fact about the use of language is that the semantical bearing of an expression is by and large a function of the context of its use. Since Frege's work in 1884[12] we have been exhorted to study the meaning of individual linguistic structures with this contextual relativity in mind. If the compositional style of semantics just discussed can be viewed as a kind of "bottom up" style of semantics — a regimen whereby the ingredients contribute to the meaning of the whole that embeds them — it is equally true that the structure of a composite whole affects the particular meanings of the ingredients. Thus pronouns in natural languages, and variables in formal systems, paradigmatically acquire what meaning they carry from the environment in which they are used.

The compositional bent of standard semantical accounts is aimed at least in part at making clear this pervasive contextual relativity. Tarski's introduction of the satisfaction relationship, for example,[13] and the ensuing ability to deal with compositional semantics of sentences formed of open as well as closed constituents, was a landmark step in formulating an explicit account of how an essentially compositional treatment could accomodate and explain the interactions among ingredients — betwen wholes and parts — that made up a particular formal system. The advent of computational formalisms has made the potential contextual dependence of particular structures more powerful and more complex: much of the debate among various proposed variable scoping protocols, and arguments for and

against side effects and global identifiers, can best be seen as having to do with the proper protocols for establishing powerful yet controlable contextualisation mechanisms.

There are a variety of techniques available for treating this contextual relativity in a formal model theory. Under one strategy — exemplified by the standard substitutional semantics for the λ-calculus, and by substitutional semantical accounts of quantification in logic — the meta-linguistic operators re-arrange the ingredients of the formal symbols so as to reduce their contextual relativity, often at the expense of a potentially infinite number of virtual expressions. In a substitutional semantical account of universal quantification, for example, a sentence "∀X[P(X)]" is taken to be true just in case all sentences of the form ⌜"P(A)"⌝ are true, where one expects A to range over *designators* of all possible objects in the semantical domain. A itself, therefore, is intended to range over *syntactic* entities. Similarly, in the λ-calculus, the term λX.F(X) is described in terms of possible substitutions into the "X" position of the body expression of all possible argument expressions. Actual applications are described in terms of particular substitutions; thus [(λX.F(X))P], for example, is taken to signify F(P).

Another strategy is to make the context of use into an explicit, reified entity, referred to by terms in the meta-language; the meaning of a contextually relative expression is then described not in terms of another possible sentence, but by making explicit reference to this theoretical posit. It is under this approach that the notion of an *environment* emerges. In such an approach to the λ-calculus example just given, for example, the body expression of the λ-term would not serve as a template for an indefinite number of substitution instances; instead, F(X) would designate the value of the function designated by the *binding of* F *in the environment of use* applied to the referent of X *again as determined by the binding in the environment of use.* Thus [(λX.F(X))P] would signify F(X) in an environment in which X was bound to whatever P designated in the environment in which the whole was being examined.

As the reader will expect, it is the latter strategy that we will adopt, both in our meta-theoretic characterisations, and in the meta-circular processors and reflective models we embed in the dialects we study. In discussing the declarative semantics of atoms, for example, we will refer to their bindings in the contextual environments; in discussing λ-abstraction and procedure bodies, we will again refer to the environments in place at the

point of use.

There are three possible confusions we need to attend to regarding the use of this notion of environment. A failure to recognise these distinctions can lead to substantial confusion later in the midst of technical details.

First, there are two different ways in which a LISP expression can depend on the context of its use: it may depend on the state of the *processor*, and it may depend on the state of the *structural field* in which it is embedded. The former we model with the notion of an environment, for although a processor state consists of both an environment and a continuation, the latter theoretical posit affects what expressions are procedurally treated, but does not itself *directly* influence the significance of a given expression. However the field (as manifested in the behaviour of CAR and CDR) can equally exert an effect as crucial as that of the environment (the binding of identifiers). We will by the term ***context*** refer to both the processor environment and the field that obtain at the point of use; the more discriminating terms will be employed when we want to refer to one or the other independently.

Second, there is a natural tendency to think that the *declarative* import of an expression would depend, to the extent that it is contextually relative, only on an environment defined by the *static* structure surrounding it as an expression. The *procedural* consequence, however, — at least so it seem at first blush — might well depend not only on the static linguistic structures surrounding it, but on the course of the computation up until the point of use.

This apparent correlation between two distinctions — procedural/declarative semantics, and static/dynamic context — is, however, ill-founded, for a variety of reasons. First, it turns out that the very notion of static environment is not without its problems: the structural field, after all, can itself be modified in the course of a computation, and only the structural field is available as a possible ground against which to *define* the notion of static context. Certainly our constant insistence on a discrimination between lexical notation and structural field implies that no dependence on *lexically* enclosing notation can possibly serve as criterial in determining the semantical import of an expression (thus we avoid the term "lexical scoping" entirely). A reflective process, clearly, can itself generate program structures which have never been notated; it can as well alter the structural field, including

the embedding structure of a program fragment. There is no way in which the context of any structural field element is irremediably frozen, immune to subsequent modification by sufficiently reflective manipuation. By *static* environment, in other words, it is not even completely clear to what we refer.

The consequences of this insight are several. First, we will *define* as ***static*** those scoping protocols that depend *only on the state of the structural field*, whereas ***dynamic*** protocols will depend by definition *on the state of the processor* (once again the processor/field distinction bears the weight of subsequent theoretical cuts). Nothing, however, will prevent us from taking the *declarative* import of symbols to depend on the *dynamic* state of the computation. Suppose I say to you that for the next five minutes we will mutually agree that each numeral will designate the number one less than that which we have always assumed. Thus during that five minute time interval we could both agree to the sentence "3 times 3 equals 5". In this way we have *dynamically* agreed to alter the undeniably *declarative* import we attribute to static expressions.

In 2-LISP and 3-LISP we will adopt a static variable scoping protocol: not because it is *necessary* in order to make sense of the notions of declarative designation, but because it facilitates the use of a base language for higher order functionality, without resort to meta-structural facilities. It is unarguably true that an additional benefit of this design choice is that the semantical import of a structural field fragment is less dependent on the course of the computation; as a consequence, for most expressions — providing no subsequent reflection alters the program structure itself — the semantical type of various variables will be readily determinable without having to determine control flow. If we were to make the other choice, however, and have declarative import determined by dynamic context, we would merely be in the familiar situation of 1-LISP, where knowledge of the state of the processor at time of use is required in order to know the semantical import of any given fragment. There is no *incoherence* in a position requiring us to know the dynamic state of the processor before being able to determine the declarative import of a structure; it is no less happy than having to know the surrounding conversation in order to determine the truth of the Perry sentence "He's *just wrong*!".

Related to this move of separating the distinction between declarative and procedural semantics from the question of the context of use, we have the question of whether *a single*

environment can be used for both the declarative and procedural semantics. Consider the following example 1-LISP fragment:

```
(LET ((VAR 6))                                          (S3-28)
    (BLOCK (SETQ VAR 7)
           VAR))
```

The question is whether this code *designates* the number six or the number seven. At first blush, it might seem that since the first form in the scope of the BLOCK operator is entirely *procedural* in nature, VAR in the last line will still have a declarative designation of six. This is, however, counter to our purposes, as the foregoing discussions imply: there is nothing conceptually incoherent about allowing SETQ to have a *dynamic* effect on the *declarative* import of subsequently interpreted structures.

Another way to put the same point is this: the *context of use* for all expressions includes both their *structural* and their *temporal* location. Declarative and procedural semantics differ on what they describe about the expression with respect to that full context: whether they describe the *designation* of the expression in the context, or whether they describe what will *happen to the symbol in virtue of being processed* in that context. In consequence, although we have a double semantics, we will maintain only a single environment structure in our meta-theory. Not only is this by far the simplest approach (any other protocol would require two different objects, a procedural environment and a declarative environment, to be handled independently throughout the meta-language characterisations), it is also the only one that coheres with intuitive practice. The natural understanding of S3-28 above is that (SETQ VAR 7) form *changes* VAR so that it henceforth (within the scope in which it is bound) *designates 7.* It is this intuition that our approach is designed to handle.

Thus our semantics is not an attempt to mitigate against practices which actually alter the meaning of extant structure: indeed, one of the demands of reflection will be to effect just such modifications to internal expressions, in a controllable way. It should be observed, however, that while it is only *procedural* import that *affects* the temporal aspect of a fragment's context, both procedural and declarative significance are thereby *affected.*

A third and final preparatory comment needs to be made regarding these environments. There are a variety of ways in which we as external theorists can treat context-relativity, even if we accept an objectified environment as part of our ontological

repertoire. The basic insight throughout the semantical treatment of LISP atoms is that an atom's designation depends on the context of evaluation. There are two ways in which we can put this precisely. We could say that an atom *has a value*, but that *that value changes in different contexts*. This way of speaking similar to how one might speak of the *value cell* of an identifier in an implementation: i.e., there is a *value cell*, and its *contents* change over the course of the computation. This, however, is an unnecessary objectification — of the space of all possible values abstracted over all possible contexts. Happier for our purposes is to make the entire notion of *having a value* itself dependent on environment, and to say that *in a given environment an atom has a particular value.*

We are using the function Ψ to relate expressions to their values: the claim, therefore, is that Ψ *is environment-relative.* In a traditional programming language semantics, the interpretation function (we will call it Γ) is always kept over-arching, so that the *meaning* (we will use the terms "meaning" and "significance" to refer to what such accounts specify of an expression, to be distinguished from what we are calling *designation*, *value*, and *procedural consequence*) of an atom (in general, of an identifier) would be said to be a function from environments to values. This technique of taking the meaning of an expression to be a complex function that incorporates the environment and state of the machine in such a way as to enable the complete articulation of the context-relativity and potential side-effects of an expression is extremely powerful, and mathematically compact, as we have pointed out before. What it should not lead us to think, however, is that the primary notions of *value* (and later, of *reference* and of *simplification*), are similarly outside the contexts of their use. An analogous situation holds regarding pronouns in natural language: in the sentence "*Bob said he would bring the ice-cream*" the pronoun "he" refers to Bob — it does not refer to a function from pragmatic and structural contexts onto objects. Rather, that function (if it could be formalised), applied in a given pragmatic and structural context, would tell you to what object, *in that pragmatic and structural context*, the pronoun refers. Though its formal embodiment would seem no more serious than to affect the order of arguments to a multi-argument function, the cost in ontological commitment is substantive.

This distinction is maintained by the mathematical vocabulary, if carefully used. The *meaning* of a variable x is taken to be a function from environments to *values*; thus the *value* is not the function itself — rather, the function takes the atom to different values,

depending on environment. The confusion arises from casual use of the natural English idiom in saying that the referent of an identifier is a "function of the environment": a phrase that is ambiguous. On one reading, it means that an identifier has a referent that is a function whose source domain is the environment; on the other, what the referent of an identifier is, is environment-relative. In order to avoid the ambiguity, since we want to take the latter reading, we will avoid the use of this apparently harmless phrase.

Our semantical functions, therefore, will be of the following type:

$$\Phi : [[\ \mathit{ENVS} \times \mathit{FIELDS}\ ] \rightarrow [\ \mathit{S} \rightarrow \mathit{D}\ ]] \qquad \text{(S3-29)}$$
$$\Psi : [[\ \mathit{ENVS} \times \mathit{FIELDS}\ ] \rightarrow [\ \mathit{S} \rightarrow \mathit{S}\ ]]$$

The field component, it is clear, will be used only by the CAR and CDR and PROP procedures; the environment component will be used by identifiers (atoms).

3.c.vi. Terminology and Standard Models

While we are on the subject of the careful use of terminology, a few additional comments should be made. First, we have been lax in our use of the terms "evaluation" and "value". In section 3.e.i we will examine this vocabulary with some care; until then we will avoid the former term, and will use "value" only with reference to a mathematical function, to refer to the element of its range for a given argument. Furthermore, it is notable that our initial analysis is of applicative calculi *in general*: we will want to talk, for example, about how bound variables are treated in the λ-calculus, in quantificational logic, in standard mathematics, in LISP 1.5, etc. We therefore cannot afford to define our analytic terms (like *binding* or *application*) with respect to any single calculus (such as for example the λ-calculus), especially since we would like these terms to support the design of new calculi satisfying some new mandates.

We will also reserve the term "function" for the mathematical abstraction, assumed to comprise an infinite set of ordered pairs in the usual fashion. By ***procedure*** we will refer to a fragment of the structural field, that we take to designate a function, and that succumbs to formal, computational treatment. (By such usage, however, we do not intend to convey the impression that a language-free — and therefore processor independent — notion of procedure should not be devised, adequate to capture what we intuitively take to be the notion of "method" or "algorithm". This is yet another point, like many in this

dissertation, where the question of the identity of a function in intension is marginally skirted.)

There is another apparently terminological issue in this vein, that hides some substantive issues regarding causal theories of reference. We have used the terms "designation" and "denotation", interchangeably, to refer to the object that a sign is taken to point to or name. We have said, for example, that the numeral 5 designates the number five, and that the symbol "Ψ" denotes the procedural processing function. All such significance is *attributed*, in the sense that the relationship between sign and signified inheres not in the sign or in the object signified, but rather in the interpreter that understands the significance relationship. This fact is recognised in standard semantics, and it is effectively admitted by the analysis that an interpreter may establish one or more of a variety of basic *models* for the signs in question. The *model theory* typically makes explicit what the designation of composite expressions is, given a basic interpretation function that takes the atomic terms onto their designata, as explained in the previous section.

It is crucial to realise that the model theory cannot *itself* specify the interpretation of a formalism, because the model theory is merely a linguistic artefact, itself in need of interpretation. Model theories are not causally grounded; they are not first-class bearers of semantic weight. Furthermore (by referring, for example, to such results as the Löwenheim-Skolem theorem) it is possible to show that any given model theoretic characterisation of a domain will admit of an infinite number of different models, all satisfying the specified constraints.

Typically, there is a *standard* model — one of a possibly infinite set of objects that everyone agrees to be the standard or default mapping of the atomic terms onto elements of an accepted semantic domain. Thus for elementary arithmetic, for example, the standard model for the signs `1`, `2`, and so forth (or, more literally, for `0`, `S(0)`, `S(S(0))`, and so forth) is the numbers as we know them, although other possible models are often explored.

There is a curious point to be made here, however, about possible models for meta-linguistic expressions. In particular, the standard model for *meta-structural* expressions *is* in fact specifiable by the model theory — there are not, in other words, indefinitely many *other* interpretations for meta-structural terms. This fact arises from the fact that *if* the model theory is admitted to be a model theory for a given set of syntactic expressions, then

it is perforce admitted to contain a number of terms that designate those syntactic expressions. Suppose in particular that some term X in the meta-language is taken to denote syntactic expression S_1 in the object language, and that some term Y denotes syntactic expression S_2. Suppose further than the meta-language posits that S_1 designates S_2, by asserting $\Phi(X) = Y$. Then it follows that S_1 must in fact designate S_2; no other interpretation is possible, since *ex hypothesi* Y is a term that refers to S_2. The freedom of interpretation inherent in the model theory, in other words, applies only to those terms not accorded meta-syntactic status by that model theory.

3.c.vii. Declarative Semantics and Assertional Force

There is a slight tendency to suppose that the suggestion that we accord LISP structures declarative semantics amounts to a suggestion that LISP be viewed as a full declarative, descriptive language. This, however, is far from the case. There are a variety of minor issues, such as that the language we are describing has variables but no quantification: such a LISP would lack, that is to say, certain kinds of expressive power (it would be what is called *algebraic*). But there is a much more serious matter that distinguishes a full fledged descriptive language from LISP: that of *assertional force.* Even with a full declarative semantics erected on the LISP field, of the sort we will depend on in 2-LISP and 3-LISP, *there is still no way to say anything!* No LISP expression can be written down with any conviction — in any way that embodies a claim. They remain detached expressions, with potentially attributed *designation*, but without any force of saying anything.

Suppose, for example, that variable X designates some atom A, and that we wish to say of atom A that it is an atom. The single argument in the redex (ATOM A), of course, does not even refer to the correct entity: it refers to whatever A designates. Rather, we would have to use (ATOM 'A). But adding this expression to the field doesn't *say* that A is an atom; rather, such a fragment could be either true or false. (ATOM '(A B C)), for example, is false, and (ATOM 'A) is true, but that fact must be determined from the outside. Nor can that fact about the truth of (ATOM 'A) itself be stated, since the problem recurses. (EQUAL T (ATOM 'A)) is as unconvincing as (ATOM 'A); it too could be true or false (we could equally well have (EQUAL T (ATOM '(A B C))). In sum, there is no mechanism — no assumption by users, and no room in the semantics — for LISP structures carrying

assertional force.

A full reflective calculus — one based on an integrated descriptive language — would have to differ in this crucial respect. Nor can one imagine this change as an addition to LISP; there is no sense in which any resultant formalism could imaginably merit the name LISP any longer, for this is a radical change. To add assertional force to LISP-like structures would be to design a fundamentally new architecture: our claim that LISP structures are best understood in terms of a declarative semantics is, rather, a reconstruction of what we claim to be present practice.

3.d. The Semantics of 1-LISP: First Attempt

The previous sections have examined what the formulation of Φ and Ψ involves in general; the discussion was not particularised to a particular dialect. In the present section we will begin to sketch those semantical functions for 1-LISP. We will at times like this dip into mathematical characterisations in order to convey a feel for how they would go, and to illustrate certain particular points. In addition, it is instructive to demonstrate the formal structure of Φ, in contrast with Ψ, since the latter function is more familiar in computational contexts (the latter, for example, is the function computed by the meta-circular processor). Nonetheless we will not present a full mathematical semantics for 1-LISP, for several reasons. First, to do so is a substantial task, well beyond the scope of this dissertation: this entire semantical analysis, it must be kept in mind, is by way of preparation for our primary investigation of reflection. Second, 1-LISP is semantically rather inelegant, and a full characterisation of it in our declarative terms would be messy, to no particular point. We will show, in particular, how an accurate account of 1-LISP's semantics would require over-riding a great many natural assumptions, in order to encode the semantically anomolous behaviour of 1-LISP's EVAL within the Φ-Ψ framework. Our goals instead are to convince the reader that such a project is at least approximately possible, to show what would be involved in doing the mathematics, and to make self-evident the truth of our main semantical result: that evaluation conflates expression simplification with term de-referencing.

Such formalisation as we do take up, will be presented in two passes. In the first, occupying the present section, we will look rather independently at the natural declarative and procedural semantics for 1-LISP; in section 3.e we will show how this approached is doomed for a variety of reasons, some stemming from peculiarities of 1-LISP's design, and some for deep reasons about the temporal aspects of any structure's context of use. In that section we will present a more complex, but more adequate, revision of the two semantical functions, with suggestions as to how complete proofs of the main results could be based on such a formulation.

3.d.i. Declarative Semantics (Φ)

We start simply, with the numerals, which designate numbers — of this there can hardly be any doubt. In addition, the two atoms T and NIL clearly signify truth and falsity. NIL is used for other purposes, of course: it is among other things an un-interpreted syntactic marker used as part of the encoding of lists within pairs, although it inherits no *designation* from that role. NIL is also the empty sequence designator, which we will take up presently.

As mentioned above, Φ is a function not only of expressions but of fields and environments; for these two simple cases, however, such context-relativisation is ignored:

∀N ∈ ***NUMERALS***, E ∈ ***ENV***, F ∈ ***FIELDS*** [ΦEF(N) = M(N)] (S3-30)

∀B ∈ ***BOOLEANS***, E ∈ ***ENV***, F ∈ ***FIELDS*** [ΦEF(B) = T(B)] (S3-31)

It follows from these equations that neither T nor NIL are available in 1-LISP for use as regular atoms — for binding, property lists, and so forth. This is false by the definition given in the last chapter, but it is true in the meta-circular processors we demonstrated in chapter 2, and it is true of most standard LISP implementations. In other words, while our *structural* characterisation made T and NIL atoms (NIL is both an atom and a list), our *procedural* taxonomies exclude it from the set of identifiers. In 2- and 3-LISP we will correct this discrepancy, having a syntactic class of two boolean constants separate from the class of atoms.

The next simplest class of structural entities are the rest of the atoms, which, from an informal point of view, are used as context-relative names. The basic intuition governing names and bound variables is this: they designate the same referent as was designated by some other expression in another environment — typically called an *argument*. Examples of this co-designative protocol can be found in both formal systems and in natural language. For example, in the sentence *"After John capsized he swam to shore."* the pronoun "he" refers to the same entity as the antecedent noun phrase "John". If another noun phrase was substituted for "John", the pronoun "he" would similarly designate that new term's referent. Thus in the sentence *"After the ragamuffin capsized he swam to shore."* "he" designates the referent of the phrase "the ragamuffin".

Similarly in the lambda calculus: in an expression of the form ((λX.<body>)E), free occurences of X in <body> are assumed, after reduction (by substitution or environment relative analysis) to designate the referent of E.

Thus the ground intuition is that the use of context-relative naming schemes provides a mechanism for establishing co-designation, in contextually dependent ways, between bound occurences of a formal name and some external expression taken up from the context. How this intuition is embodied in the formal treatment is open to a variety of alternatives: in the λ-calculus, for example, as we discussed in section 3.c.v, no notion of environment is required: instead a full substitutional protocol is adopted in which the contextual term is *substituted* for the appropriate occurrences of the variable within the expression in question. For compatibility with our theoretical reconstructions of dynamic scoping protocols, however, and in order to establish close alignment between our meta-theoretical accounts and our subsequent reflective models, we will adopt the theoretical posit of an environment as an explanatory mechanism with which to explain this contextual relativisation of variable designations.

We will not, however, adopt a notion of *value* with respect to variables, because of the use/mention confusions that attend common use of that term (see section 3.f.i). The problem, in a word, is whether the value of a variable or formal parameter is taken to be the argument expression itself, or the *referent* of the argument expression (by *argument* we refer to the contextually determined expression with which the variable is assumed to be co-designative). For example, in the following two expressions, there can be no doubt, in the contexts in which (+ X 1) has its intended meaning, that the variable X is intended in each case to designate the *number* four.

```
((LAMBDA (X) (+ X 1)) 4)                                    (S3-32)
((LAMBDA (X) (+ X 1)) (+ 2 2))
```

We will, however, need to decide what sort of entity the environment establishes as the ***binding*** of a variable. The question is whether, in the environments established by the applications just illustrated, the variable X is bound to the actual number four, or to a *designator* of that number. This question is independent of the clear fact (this is true in mathematics and logic as well, giving us some confidence) that, semantically, variable binding is co-referential in the sense that the variable, in virtue of being bound to an argument expression, acquires the designation *of that argument.*

If we are to write down Φ for atoms — LISP's variables — we have to make one decision or the other. The two candidate equations are these (assuming, as we will throughout, that environments are functions from atoms to bindings — i.e., that E : *[ATOMS → D]* in S3-33 and that E : *[ATOMS → S]* in S3-34):

$$\forall A \in \mathit{ATOMS},\ E \in \mathit{ENV},\ F \in \mathit{FIELDS}\ \ [\Phi EF(A) = E(A)] \qquad \text{(S3-33)}$$

$$\forall A \in \mathit{ATOMS},\ E \in \mathit{ENV},\ F \in \mathit{FIELDS}\ \ [\Phi EF(A) = \Phi EF(E(A))] \qquad \text{(S3-34)}$$

An apparent argument for the first option (S3-33 — that bindings are designations) is the fact that variable binding as normally conceived is extensional, and furthermore, that the expression to which variable is bound is not itself normally thought to be preserved in the binding. It would seem, if the second proposal were adopted, that the only natural expression to which the variable should be bound is the one occuring in context when the binding takes place (i.e., 4 or (+ 2 2) in the examples in S3-32 above), and this is certainly not how binding is presumed to work. In fact S3-34 has the odd consequence that in any environment the designation of the binding of a variable (not what the variable is bound to, but what entity is the referent of the expression *that* the variable is bound to) is potentially a function of the environment *in which the variable is itself used or looked up* (this is because the outer term of S3-34 is ΦEF(...)). This would seem wildly counter-intuitive.

On the other hand, arguing for the second alternative (S3-34 — that bindings are expressions) is the fact that under the first alternative the bindings of 1-LISP variables will not in general be s-expressions. This is exactly the extensional point just made: if we adopt S3-33, we would say that X was bound to the *number four*, not to the numeral "4". This is not a problem in the meta-language, but it makes for odd consequences for the meta-circular processor (and later for reflective machinations). No environment, in other words, can be a LISP object, and (EVAL X) will not be able to return X's binding.

A possible reply to this last objection is that we would not expect environments *themselves* to consist of pairs of s-expressions: rather, the only LISP structure we would likely want is a structural *designator* of an environment. Thus if X were bound to the *number* four, then a sequence of two designators, one designating X and the other designating four, would serve as the environment *structure*. The only difficulty with this counter suggestion is that those designators might themselves be environment relative: if X were bound to four, the environment designator might consist of the tuple 'X ('X, as we

will see, is a natural designator of the atom X) and the expression (+ Y Y), if Y designates two. This would seem an unhappy pratice. It seems not unreasonable to require that environments encode the full context-relativity of a variable's binding, rather than simply deferring it onto another context.

Furthermore, against the objection that (EVAL X) cannot return X's binding we have the following rejoinder: there is a mistake in the intuition that (EVAL X) should return X's binding, *if binding is taken to be designational.* EVAL is LISP's Ψ: even if EVAL is declaratively extensional, we would expect (EVAL X) to designate the *procedural consequence* of the referent of X, not the declarative import of the referent of X. ΦEF("*(EVAL X)*), in other words, should be ΨEF(ΦEF("*X*)), not ΦEF(ΦEF("*X*)).

We find ourselves in the thick of issue discussed in section 3.c.v, in which the context relativity of both declarative and procedural import come into tension. This last discussion of the proper designation (EVAL X) brings to the fore the question of whether the declarative and procedural environments can be considered to be the same. It is clear — since Ψ maps structures onto structures — that *from a procedural point of view* the environment *cannot* be the first, designational, alternative. If there is any hope of letting a single theoretical entity serve a double role as both procedural and declarative context, then, we would have to choose the second of the two alternatives.

In sum, the first option, by which bindings are designative, is coherent, although it is affected by two complications:

1. LISP encodings of environments may use context-relative designators of the bindings;
2. Bindings so construed cannot be taken to be the procedural consequence of variables. Ψ of a variable, in other words, cannot be its binding, on this reading.

The second alternative, by which bindings are co-designative, has in contrast the following apparent difficulty:

1. It is unclear what expression the binding should be: the contextually relative argument expression means that the semantics of the binding is potentially a function of the environment at the point where the variable was bound.

It might seem that the environment could "record" the context in which the binding took place, so that instead of the designation of a variable being ΦEF(E(A)), it would be

ΦE*F*(E(A)), where E* is the environment at the point of binding, and F* the state of the field. This, however, is an empty proposal: it is effectively indistinguishable in effect from the first, except more complexly formulated.

In fact there is a third option: variables could be taken to be bound to co-designative expressions, but *not to the expression occuring in the binding context.* In standard LISPs no such expression presents itself, but in 2-LISP we will posit that variables are bound instead to a *normal-form* expression having the same referent as that of the primary argument. This avoids the trouble just discussed, because in those dialects all normal-form designators are *context-independent* (in terms of declarative designation); hence the additional context arguments to Φ in S3-34 are provably ignored (being required only to satisfy the category requriements of the meta-theoretic characterisation). Thus in those dialects we will adopt the second equation without difficulty. However it would be premature to adopt this suggestion yet: we haven't yet defined normal-form designators, and to make this suggestion work we have to *prove* that they are environment independent, and so forth.

Nonetheless, the mandate adopted in 3.c.v requires that a single theoretical posit serve as both declarative and procedural environment; this requirement alone rejects the first, designational, alternative. What we will adopt is the following rather mixed protocol: we will assume that 1-LISP variables are bound to *some* expression, and we will merely assert, in the axiomatisation of the declarative semantics, the declarative import of the binding. Any choice of binding satisfying the equations will be accepted as valid, from the point of view of the declarative semantics; thus for example a regimen that identified a particular special symbol, one per object in the semantical domain, would suffice. When we get to the 1-LISP procedural semantics we will make plain what object is in fact bound to each variable; when we turn to 2-LISP we will defend that dialect's choice of such an object on semantical grounds.

We will therefore proceed under the second equation, by which environments are taken in the meta-language to be functions from variables to expressions co-designative with the argument expressions. Thus we are adopting:

∀A ∈ ***ATOMS***, E ∈ ***ENV***, F ∈ ***FIELDS*** [ΦEF(A) = ΦEF(E(A))] (S3-35)

We have discussed the booleans T and NIL, the numerals, and the atoms in general. There are two further categories of symbol to look at, before turning to compositional questions: the bindings of the primitive atoms in the initial environment, and the designation of pairs. Since all primitive atoms are bound to procedures (i.e., all twenty-nine atoms that *have* bindings in the initial context are bound to procedures), and since the semantical import of procedures is best revealed in terms of their participation in the encoding of "function applications", we will turn to pairs next.

There is a choice here: as noted in the previous chapter, 1-LISP differs substantially from 1.7-LISP; the latter evaluating the first position in a function application designator in the standard sense. Because 1.7-LISP is closer in spirit to the later dialects we will deal with, and because it is more general than 1-LISP, we will consider it.

Pairs, of course, are not quite the right category to examine: we want instead to focus on *lists*. The simplest suggestion for the designation of a list (those, at least, used to signify function applications, rather than those used as enumerators), is this: a *list* will be taken to designate the value of the function designated by its first element applied to the arguments designated by its remaining elements:

$$\forall S \in \mathcal{S},\ E \in \mathit{ENV},\ F \in \mathit{FIELDS} \qquad \text{(S3-36)}$$
$$[\Phi E_F(S) = \Phi EF(S_1)\ (\langle \Phi EF(S_2), \Phi EF(S_3), \ldots \Phi EF(S_k)\rangle)]$$
$$\text{if } S = \ulcorner(\underline{S_1}\ \underline{S_2}\ \underline{S_3}\ \ldots\ \underline{S_k})\urcorner$$

where by $S = \ulcorner(\underline{S_1}\ \underline{S_2}\ \underline{S_3}\ \ldots\ \underline{S_k})\urcorner$ we will in general mean:

$$[[\,\mathrm{CAR}(S) = S_1\,] \wedge [\,\mathrm{CAR}(\mathrm{CDR}(S)) = S_2\,] \wedge \ldots \qquad \text{(S3-37)}$$
$$\wedge\ [\,\mathrm{CAR}(\mathrm{CDR}(\mathrm{CDR}\ \ldots\ (\mathrm{CDR}(S))\ldots))) = S_k\,]]$$

or more precisely, since CAR is not a function in the meta-language (recall that by F^i we in general mean the i'th element of sequence F; thus, since fields are of type $[\mathit{CARS} \times \mathit{CDRS} \times \mathit{PROPS}]$, F^1 is the CAR relationship of field F):

$$[[\,F^1(S) = S_1\,] \wedge [\,F^1(F^2(S)) = S_2\,] \wedge \ldots \qquad \text{(S3-38)}$$
$$\wedge\ [\,F^1(F^2(F^2\ \ldots\ (F^2(S))\ldots))) = S_k\,]]$$

This is just the sort of semantical equation for applications one would expect in any semantical treatment; an example will illustrate. Suppose we inquire about the designation of the expression (+ 3 Y) in an environment E_0 in which Y is bound to a designator of the number four, and + is bound to a designator of the addition function. We would have the following (as discussed in section 3.c.iv, we use a single double quote mark and an italic

font to mention object-level structures; all non-italicised items, such as "+" and "7", are terms in the meta-language):

$$\Phi E_0F_0(\text{"}(+\ 3\ Y))$$ (S3-39)
$$= [(\lambda E.\lambda F.\lambda S\ \Phi EF(S_1)\ [\Phi EF(S_2),\ \Phi EF(S_3),\ \ldots\ \Phi EF(S_k)])E_0F_0][\text{"}(+\ 3\ Y)]$$
$$= [(\lambda F.\lambda S\ \Phi E_0F(S_1)\ [\Phi E_0F(S_2),\ \Phi E_0F(S_3),\ \ldots\ \Phi E_0F(S_k)])F_0][\text{"}(+\ 3\ Y)]$$
$$= (\lambda S\ \Phi E_0F_0(S_1)\ [\Phi E_0F_0(S_2),\ \Phi E_0F_0(S_3),\ \ldots\ \Phi E_0F_0(S_k)])[\text{"}(+\ 3\ Y)]$$
$$= \Phi E_0F_0(\text{"}+)\ [\Phi E_0F_0(\text{"}3),\ \Phi E_0F_0(\text{"}Y)]$$
$$= \Phi E_0F_0(E_0(\text{"}+))\ [\Phi E_0F_0(\text{"}3),\ \Phi E_0F_0(\text{"}Y)]$$
$$= +[N(\text{"}3),\ \Phi E_0F_0(E_0(\text{"}Y))]$$
$$= +[3,4]$$
$$= 7$$

The importance of S3-39 is, in line with our general conception of Φ, to indicate that the expression (+ 3 Y) designates seven in an environment in which Y is bound to four, *still apart from any notion of how it is to be treated by the processor.* It is to be noted, for example, that the expression designates an abstract number, not the *numeral* 7, which has not once been mentioned in this analysis. Only when we describe the procedural treatment Ψ of (+ 3 4) will the numeral 7 enter into the analysis.

Two comments in passing. The first is terminological: the term (+ 3 Y) *designates* seven; therefore we cannot strictly say that it designates an *application* of the addition function to 3 and Y. Lists, in other words, cannot be said to *designate* function applications. On the other hand, pairs are not *themselves* function applications either, since the CAR of the list, for example, is a function designator, not a function. We will explore the language of functions and applications more fully in section 3.f.i; for the time being we will call lists of this variety (i.e., lists whose significance is explained in terms of the application of the designation of their first element to the arguments encoded in the rest of the list) ***procedure applications***, although after the discussion in section 3.f.i we will replace the term "application" with "reduction". Although we are not dealing in this dissertation with notions of intension, what we would ultimately like to say is that the intension of a procedure application is a function application; the extension is the value of the function applied to the arguments. In deference to such a wish, and in what must for now remain a rather informal usage, we will sometimes say that lists *signify* function applications.

The second comment is this: As is clear from the examples, we are using an extended version of the lambda calculus with identifiers as our meta-language. Note that in S3-39 we expanded the composite term in the first line under a "substitution semantics"

regime: occurences of the bound variable E were replaced by the *term* E_0. The resulting expression is of course still extensional in that position into which E_0 was substituted. 1-LISP of course would *evaluate* — whatever that means — the term E_0 before evaluating the body of the procedure. It should be clear from the fact that our meta-language is well-formed that there is no need for an evaluation process to de-reference the argument, as LISP's EVAL is sometimes thought to do, in order for composite appliications to be extensional. There may be other reasons for "evaluating" the arguments to a procedure — in fact there are several, as we will see — but a need to *de-reference*, as this example shows, is not one of them.

To return to the main discussion, we must acknowledge an inconsistency in the account we have given: we have said that the designation of lists is the value of the function designated by the first element applied to the designations of the remaining elements. But tails of lists in 1-LISP are themselves lists; a strict reading of our analysis would imply not only that $\Phi EF("(PLUS\ 3\ 4)) = 7$, but also that $\Phi EF("(3\ 4))$ designated the value of the function designated by the numeral 3 applied to the number four. However there is no such value: the numeral 3 designates (by S3-30) the number three, which is not a function at all. We could define Φ to take (3 4) into $\perp$, or into an *error*, but to do so would be to begin to let Φ drift away from our lay understanding. The expression (3 4) is not a functional term to us, and therefore we should not let our semantical characterisation treat it as one. In point of fact, of course, it designates a *sequence of integers*, a semantic import conveyed by the following semantical equation:

$$\forall S \in \mathbf{S},\ E \in \mathbf{ENV},\ F \in \mathbf{FIELDS} \qquad \text{(S3-40)}$$
$$[\Phi EF(S) = \langle \Phi EF(S_1),\ \Phi EF(S_2),\ \Phi EF(S_3),\ \ldots\ \Phi EF(S_k)\rangle]$$
$$\text{if } S = \ulcorner"(\underline{S_1}\ \underline{S_2}\ \underline{S_3}\ \ldots\ \underline{S_k})\urcorner$$

In order to know when a list is intended to designate a sequence, however, we need to know the context it appears in — or the contexts, since a given s-expression can occur as the ingredient in more than one larger expression. Such a move, however, entails violating recursive compositionality of the semantics, which is highly inelegant in a formal system.

These troubles are merely evidence of the lack of type-type correspondence, made explicit in section 1.f.i, between the syntactic categorization of the structural field *S* and its semantical interpretation. We could try to complicate our definition of Φ so as to restrict its application to lists which really *were* intended to signify function applications, but this is

of course impossible: *intention* is not something a formally defined procedure can unravel. The consequence is not minor: for 1-LISP it is in general impossible to tell *syntactically* what the *semantic type* of an s-expression is (or even whether it bears semantic weight). We can never require that it be possible to tell syntactically what every expression's semantical *import* is: for all formalisms of any substantial power such a question is intractable. However requiring that structures wear their semantic *category* — not the category of the *referent*, but rather the category into which the semantical function Φ sorts syntactic entities — on their sleeve is neither an impossible nor an unreasonable requirement. Again, this is an inelegance we will correct in 2- and 3-LISP.

The foregoing extensional reading of procedure applications will fail when we get to LISP's so-called *special forms*; before revising it in order to handle them, however, we can look at some of the standard LISP extensional primitives.

There are twenty-three distinguished atoms in 1-LISP; of these we have already given the semantics of T and NIL. Three others (QUOTE, LAMBDA, and COND) will be dealt with separately in a moment, and four more (SET, DEFINE, READ, and PRINT) are significant primarily procedurally, so will be discussed later. Finally, EVAL and APPLY — of particularly importance in our overall drive for reflection, which is motivating all of this semantical analysis — will receive special attention later. Of the remaining twelve, ten would have the following designations in the initial environment E_0 and the initial field F_0. (Note that we use $\Phi E_0F_0("X)$ rather than the equivalent but more cumbersome $\Phi E_0F_0(E_0("X))$.)

$\Phi E_0F_0("CAR) = F_0^1$; since F = <CAR_0, CDR_0, $PROP_0$> (S3-41)
$\Phi E_0F_0("CDR) = F_0^2$; similarly
$\Phi E_0F_0("PROP) = F_0^3$; similarly
$\Phi E_0F_0("EQ) = \lambda\langle S_1,S_2\rangle . [S_1 = S_2]$
$\Phi E_0F_0("+) = +$
$\Phi E_0F_0("-) = -$
$\Phi E_0F_0("*) = *$
$\Phi E_0F_0("/) = /$
$\Phi E_0F_0("NUMBERP) = \lambda S . [S \in \mathit{INTEGERS}]$
$\Phi E_0F_0("ATOM) = \lambda S . [S \in \mathit{ATOMS}]$

Five of these functions are effectively *absorbed* in our meta-langauge, in the sense that the same concept is used in the meta-language as is being explained in the object language; thus these semantical characterisations are not illuminating. (Though the term is ours, the practice is not: conjunction, for example, is typically absorbed in a first-order semantics, since $\ulcorner P \wedge Q \urcorner$ is said to be true just is case P is true *and* Q is true. This is analogous to the

use of the term "reflection" in logic's *reflection principles*, although we of course must avoid that term in this context.) Axioms constraining them could of course be formulated, but since our goal is to indicate a *style* of semantical analysis, not to actually lay out a valid 1-LISP semantics, we will simply assume that these functions are clearly defined. Two others are simply simple type predicates designating truth or falsity depending on the designations of their arguments. Finally, three (CAR, CDR, and PROP) are simply the relationships extracted from the FIELD argument to Φ; these in fact are the only procedures that access that crucial constituent in describing the field. Note that none of these procedures need to "de-reference" their arguments, as that task is performed in general by the semantics of applications, as stated in S3-36, and as illustrated in the example in S3-39.

We look next at what in the community are sometimes called *special forms*: lists whose first element designates something other than an extensional function. There are a variety of such forms, and two ways in which we could proceed to analyse them. The first, represented by the first meta-circular 1-LISP interpreter we demonstrated in chapter 2, is to consider a certain number of atoms as *specially marked*, and to make explicit what applications formed in terms of them designate. The second, which we adopted in our second meta-circular interpreter, is to identify a special class of procedures (called FEXPRS in MACLISP and NLAMBDAS in INTERLISP — in 3-LISP they will be subsumed by the general class of reflective procedures; for the present we will call them *intensional* procedures). Since this is both cleaner and will put us in a better position to handle subsequent developments, we will adopt this latter stance, and first lay out a protocol for dealing with intensional procedures in general, and then subsequently define the particular semantics of the three primitive intensional procedures QUOTE, LAMBDA, and COND.

The problem with intensional procedures is of course that applications formed in terms of them, such as (QUOTE HELLO) or (LAMBDA (X) (+ X 1)), do not satisfy the mandate laid down in S3-36 claiming that their designation is the value of the function designated by the first element of the list applied to the designations of the remaining elements (i.e., to use INTERLISP terminology, LAMBDA is an NLAMBDA). In particular:

$$[\Phi E_0 F_0(\text{"}(QUOTE\ HELLO)\text{"})] \neq [(\Phi E_0 F_0(\text{"}QUOTE\text{"}))\ [\Phi E_0 F_0(\text{"}HELLO\text{"})]] \qquad \text{(S3-42)}$$

$$[\Phi E_0 F_0(\text{"}(LAMBDA\ (X)\ (+\ X\ 1))\text{"})] \neq [(\Phi E_0 F_0(\text{"}LAMBDA\text{"}))\ [\Phi E_0 F_0(\text{"}(X)\text{"}).\Phi E_0 F_0(\text{"}(+\ X\ 1))\text{"})]] \qquad \text{(S3-43)}$$

A candidate solution would be to rework S3-36 so as not to de-reference its arguments, and then to redefine the functions designated by the atoms CAR, CDR, and so forth, to make this move explicitly. Then applications in *general* will not be extensional; only those we explicitly indicate as extensional will be so. We would also have to redefine these functions to accept the environments as an explicit argument, so that they themselves can de-reference their arguments when appropriate. Thus we would have (we will cease explicitly identifying the category restrictions on S, E, and F):

$$\forall S \in \mathit{S},\ E \in \mathit{ENVS},\ F \in \mathit{FIELDS} \qquad \text{(S3-44)}$$
$$[\Phi EF(S) = [(\Phi EF(S_1))EF]\ \langle S_2, S_3, \ldots S_k\rangle]$$
$$\text{if } S = \ulcorner\text{"}(\underline{S_1}\ \underline{S_2}\ \underline{S_3}\ \ldots\ \underline{S_k})\text{"}\urcorner$$

Our primitive functions would have to be redefined appropriately. As an example, the atom +, under this approach, would have the following designation in the initial environment:

$$\Phi E_0F_0(E_0(\text{"+})) = \lambda E.\lambda F.\lambda X,Y\ +[\Phi EF(X),\ \Phi EF(Y)] \qquad \text{(S3-45)}$$

That this would be correct is shown by redoing the example of S3-39:

$$\begin{aligned}\Phi E_0F_0(\text{"(+ 3 Y)}) &= [(\Phi E_0F_0(\text{"+}))E_0F_0]\ (\text{"3},\ \text{"Y}))\\ &= [\Phi E_0F_0(E_0(\text{"+}))E_0F_0]\ (\text{"3},\ \text{"Y})\\ &= [(\lambda E.\lambda F.\lambda X,Y\ +(\Phi EF(X),\ \Phi EF(Y)))E_0F_0]\ (\text{"3},\ \text{"Y})\\ &= [(\lambda F.\lambda X,Y\ +(\Phi E_0F(X),\ \Phi E_0F(Y)))F_0]\ (\text{"3},\ \text{"Y})\\ &= [\lambda X,Y\ +(\Phi E_0F_0(X),\ \Phi E_0F_0(Y))]\ (\text{"3},\ \text{"Y})\\ &= +(\Phi E_0F_0(\text{"3}),\ \Phi E_0F_0(\text{"Y}))\\ &= +(3,\ \Phi E_0F_0(E_0(\text{"Y})))\\ &= +(3,\ 4)\\ &= 7\end{aligned} \qquad \text{(S3-46)}$$

In an analogous fashion we could redefine the other primitives of S3-41:

$$\begin{aligned}\Phi E_0F_0(\text{"}CAR) &= \lambda E.\lambda F.\lambda X\ .\ [F^1(\Phi EF(X))]\\ \Phi E_0F_0(\text{"}CDR) &= \lambda E.\lambda F.\lambda X\ .\ [F^2(\Phi EF(X))]\\ \Phi E_0F_0(\text{"}PROP) &= \lambda E.\lambda F.\lambda X\ .\ [F^3(\Phi EF(X))]\\ \Phi E_0F_0(\text{"}EQ) &= \lambda E.\lambda F.\lambda X,Y\ .\ [\Phi EF(X) = \Phi EF(Y)]\\ \Phi E_0F_0(\text{"-}) &= \lambda E.\lambda F.\lambda X,Y\ .\ [-(\Phi EF(X),\ \Phi EF(Y))]\\ \Phi E_0F_0(\text{"*}) &= \lambda E.\lambda F.\lambda X,Y\ .\ [*(\Phi EF(X),\ \Phi EF(Y))]\\ \Phi E_0F_0(\text{"/}) &= \lambda E.\lambda F.\lambda X,Y\ .\ [/(\Phi EF(X),\ \Phi EF(Y))]\\ \Phi E_0F_0(\text{"}NUMBERP) &= \lambda E.\lambda F.\lambda X\ .\ [\Phi EF(X) \in \mathit{INTEGERS}]\\ \Phi E_0F_0(\text{"}ATOM) &= \lambda E.\lambda F.\lambda X\ .\ [\Phi EF(X) \in \mathit{ATOMS}]\end{aligned} \qquad \text{(S3-47)}$$

Given this change in approach, we could begin to define some intensional procedures. First we take the atom QUOTE, which clearly designates the name of its argument. In other words, for all expressions X we will require that $\Phi EF(\text{"}(QUOTE\ X)) = \text{"}X$:

$$\Phi E_0F_0(\text{"}QUOTE) = \lambda E.\lambda F.\lambda X.X \qquad \text{(S3-48)}$$

Given this equation, we can show how the structure (QUOTE (THIS IS A LIST)) designates the list (THIS IS A LIST), in all environments in which QUOTE has this meaning:

$$\begin{aligned}&\forall E \in \mathit{ENVS},\ F \in \mathit{FIELDS} \qquad \text{(S3-49)}\\ &\quad [\![\ \Phi EF(\text{"}QUOTE) = \lambda E.\lambda F.\lambda X.X\ ] \supset\\ &\qquad [\![\ \Phi EF(\text{"}(QUOTE\ (THIS\ IS\ A\ LIST)))\ ]\\ &\qquad\quad = [\Phi EF(\text{"}QUOTE)EF]\ (\text{"}(THIS\ IS\ A\ LIST))\\ &\qquad\quad = [(\lambda E.\lambda F.\lambda X.X)EF]\ (\text{"}(THIS\ IS\ A\ LIST))\\ &\qquad\quad = [(\lambda F.\lambda X.X)F]\ (\text{"}(THIS\ IS\ A\ LIST))\\ &\qquad\quad = [\lambda X.X]\ (\text{"}(THIS\ IS\ A\ LIST))\ \ ;\ \text{The context is thrown away}\\ &\qquad\quad = \text{"}(THIS\ IS\ A\ LIST)\ ]\!]\end{aligned}$$

Since this derivation makes no claim upon the structure of its argument, it can be generalised to the following theorem:

$$\forall S \in \mathcal{S}.\ \forall E \in \textit{ENV}.\ F \in \textit{FIELDS} \qquad \text{(S3-60)}$$
$$[[\Phi EF(\text{"}\textit{QUOTE}) = \lambda E.\lambda F.\lambda X.X] \supset [\Phi EF(\ulcorner\text{"}(\textit{QUOTE}\ \underline{S})\urcorner) = S]]$$

Note that we have quite reasonably assumed that the LISP operator "QUOTE" designates the hyper-intensional identity function. *It should be absolutely clear* that this definition of "QUOTE" makes no reference at all to any concept of evaluation, an issue we have not yet considered. It will be a matter of some interest to see, once we have characterized LISP's notion of evaluation in terms of the semantical framework we are currently erecting, whether the manner in which "QUOTE" is handled by the interpreter is consonant with the definition just articulated.

During all of this discussion we have used the subjunctive; the problem is that in spite of its increased power there is something inelegant about this move of having all function designators designate intensional functions. Note that we have now said that the atom "+" does not designate the addition function: rather, it designates a function from contexts to a function from structures to numbers — i.e., it is of type $[[\textit{ENVS} \times \textit{FIELDS}] \rightarrow [\mathcal{S} \rightarrow \textit{INTEGERS}]]$. A certain amount of "semantic innocence" has been lost in making the simple procedures complex, in order to make more complex procedures simple.

Furthermore, this approach is too general: it allows us to posit, as the designation of 1-LISP procedures, functions with arbitrary "de-referencing" power, whereas in fact 1-LISP procedures must be of only two varieties: those that are extensional in their arguments (EXPRS), and these that are not (IMPRS); there is no way to define a 1-LISP procedure of *intermediate* extensionality (one that de-references just one of its two arguments, for example).

A cleaner strategy, it would seem, would be to define a meta-linguistic predicate, called, say, EXT?, which was true of extensional functions and false of intensional ones. If we could do that, we could recast the declarative semantics of lists as follows, without giving the *intensional* functions the environment as an argument, thus preventing them from de-referencing *any* of their arguments:

$$\forall S \in \mathcal{S}.\ E \in \textit{ENVS}.\ F \in \textit{FIELDS} \qquad \text{(S3-61)}$$
$$\Phi EF(S) = \textit{if}\ EXT?(\Phi EF(S))$$
$$\textit{then}\ \Phi E^{\cdot}(S_1)\ [\Phi EF(S_2), \Phi EF(S_3), \ldots \Phi EF(S_k)]$$

$$\textit{else } \Phi EF(S_1)\ [S_2, S_3, \ldots S_k]$$
$$\textit{if } S = \ulcorner *(\underline{S_1}\ \underline{S_2}\ \underline{S_3} \ldots \underline{S_k})\urcorner$$

The problem, however, is that such a predicate (EXT?) is impossible. The difficulty is illustrated by the following:

```
(DEFINE F1 (LAMBDA EXPR (A) (CAR A)))                    (S3-62)
(DEFINE F2 (LAMBDA IMPR (A) (CAR A)))
```

For example, we would have the following behaviour:

```
(F1 (CONS 3 4))      →   3                               (S3-63)
(F2 (CONS 3 4))      →   CONS
```

Under the treatment suggested in S3-61 above, both F_1 and F_2 would be required to have the same denotation; in particular, $\Phi EF(F_1)$ and $\Phi EF(F_2)$ would both have to be the CAR function. Since they are identical, there is therefore no way in which (EXT? F_1) can be true and (EXT? F_2) be false. Another way to see this is to realise that, in spite of our use of what is common terminology, it is not *functions* that are intensional or extensional; rather it is only to *procedures* (or to some other more intensional object) that we can properly apply these terms.

For these reasons we will adopt a third possibility — one that in the meta-theoretic language maintains the clarity of our first suggestion, that adequately treats INPRS, and that does not provide as much generality as the option just explored. The approach is to mediate between the two previous proposals, as follows. First we define the following two meta-linguistic functions, which we call the ***extensionalisation*** and ***intensionalisation*** functions (these can be understood as the designational analogues of the procedural EVLIS in McCarthy's original report[14]):

$$EXT \equiv \lambda G.\lambda E.\lambda F.\lambda S\ .\ G[\Phi EF(S_1), \Phi EF(S_2), \ldots \Phi EF(S_k)] \qquad \text{(S3-64)}$$
$$\text{where } S = \ulcorner *(\underline{S_1}\ \underline{S_2}\ \underline{S_3} \ldots \underline{S_k})\urcorner$$

$$INT \equiv \lambda G.\lambda E.\lambda F.\lambda S\ .\ G[S_1, S_2, \ldots S_k] \qquad \text{(S3-65)}$$
$$\text{where } S = \ulcorner *(\underline{S_1}\ \underline{S_2}\ \underline{S_3} \ldots \underline{S_k})\urcorner$$

EXT is a *functional*: a function defined over other functions, that transforms them into functions that pick up an environment and de-reference the arguments first, and then apply the original function to the resulting referents. INT, by contrast, transforms a function into functions that pick up an environment but ignore it, applying the original function to the arguments as is. We can now say that in E_0 the atom "+" designates EXT(+), where + in the

latter term is the meta-linguistic name for the real addition function. QUOTE, on the other hand, designates INT(λY.Y). We will then *require*, as a meta-linguistic convention, that all function designators be restricted to those built from EXT or INT. These are both straightforward and perspicuous, as is the new (recursive) definition of Φ (we show just the fragment for pairs):

$$\Phi \equiv \lambda E.\lambda F.\lambda S.\ [(\Phi EF(S_1))EF]\ (S_2) \qquad \text{(S3-56)}$$
where $S = \ulcorner"(\underline{S_1} . \underline{S_2})\urcorner$

That this works is shown by the following two examples:

$\Phi E_1F_1("(+\ 3\ Y))$; where Y designates 4 in E_1 (S3-57)
$= [(\Phi E_1F_1("+))E_1F_1]\ "(3\ Y)$
$= [(EXT(+))E_1F_1]\ "(3\ Y)$
$= [((\lambda G.\lambda E.\lambda F.\lambda S\ G[\Phi EF(S_1), \Phi EF(S_2), \ldots \Phi EF(S_k)])+)E_1F_1]\ "(3\ Y)$
$= [(\lambda E.\lambda F.\lambda S\ +[\Phi EF(S_1), \Phi EF(S_2)])E_1F_1]\ "(3\ Y)$
$= [(\lambda F.\lambda S\ +[\Phi E_1F(S_1), \Phi E_1F(S_2)])F_1]\ "(3\ Y)$
$= [\lambda S\ +[\Phi E_1F_1(S_1), \Phi E_1F_1(S_2)]]\ "(3\ Y)$
$= +[\Phi E_1F_1("3), \Phi E_1F_1("Y)]$
$= +[3, \Phi E_1F_1(E_1("Y))]$
$= +[3, 4]$
$= 7$

and:

$\Phi E_1F_1("(QUOTE\ (HELLO\ THERE)))$ (S3-58)
$= [(\Phi E_1F_1("QUOTE))E_1F_1]\ "((HELLO\ THERE))$
$= [(INT(\lambda X.X))E_1F_1]\ "((HELLO\ THERE))$
$= [((\lambda G.\lambda E.\lambda F.\lambda S\ \ G[S_1, S_2, \ldots S_k])\lambda X.X)E_1F_1]\ "((HELLO\ THERE))$
$= [(\lambda E.\lambda F.\lambda S\ (\lambda X.X)[S_1])E_1F_1]\ "((HELLO\ THERE))$
$= [(\lambda F.\lambda S\ (\lambda X.X)[S_1])F_1]\ "((HELLO\ THERE))$
$= [(\lambda S\ (\lambda X.X)(S_1))]\ "((HELLO\ THERE))$
$= (\lambda X.X)\ "(HELLO\ THERE)$
$= "(HELLO\ THERE)$

Note how in the third from last line the environment E_1, which has been carefully passed in to the function, is ignored by the "intensionalised" function.

In other words, the new Φ of S3-56 is adequate for both extensional and intensional procedures, which is what we wanted of it. It is also meta-mathematically perspicuous, and it is of just the right power. Accordingly, we can now set down the equations that must be satisfied by the initial environment E_0 for the primitive procedures we have looked at so far. (Note that CAR and CDR cannot be defined in terms of EXT, even though they are extensional, because they need access to the : $EXT(F^1)$ is ill-formed since F is not bound.) We will not consider this a violation of our convention, however, since they are in fact still

extensional in the sense that they designate functions of the extensions of their arguments.)

$\Phi E_0F_0("CAR) = \lambda E.\lambda F.\lambda X\ .\ [F^1(\Phi EF(X))]$ (S3-59)
$\Phi E_0F_0("CDR) = \lambda E.\lambda F.\lambda X\ .\ [F^2(\Phi EF(X))]$
$\Phi E_0F_0("PROF) = \lambda E.\lambda F.\lambda X\ .\ [F^3(\Phi EF(X))$
$\Phi E_0F_0("+) = EXT(+)$
$\Phi E_0F_0("-) = EXT(-)$
$\Phi E_0F_0("*) = EXT(*)$
$\Phi E_0F_0("/) = EXT(/)$
$\Phi E_0F_0("EQ) = EXT(=)$
$\Phi E_0F_0("NUMBERP) = EXT(\lambda S.S \in INTEGERS)$
$\Phi E_0F_0("ATOM) = EXT(\lambda S.S \in ATOMS)$
$\Phi E_0F_0("COND) = EXT(\lambda X.$ *if* $X_{1,1}$ *then* $X_{1,2}$
elseif $X_{2,1}$ *then* $X_{2,2}$ ... etc.
where $X = \Gamma"((X_{1,1}\ X_{1,2})\ (X_{2,1}\ X_{2,2})\ \ldots)\urcorner$
$\Phi E_0F_0("QUOTE) = INT(\lambda S.S)$

There are several comments to be made about this list. First, note that COND is described as an *extensional* procedure, declaratively: this is correct — COND will be shown to be *procedurally* intensional, because it evaluates its arguments in normal, rather than applicative, order. From a declarative point of view, however, the designation of a COND application is a function only of the referents of its arguments (as of course are "*if* ... *then* ... *else* ... " and the material conditional in the meta-language).

Two procedures that are important, but simply described, are EVAL and APPLY. As one might expect, the natural reading of the designation of an application formed in terms of EVAL is that it designates the procedural consequence of the referent of its argument. Thus for example we expect (EVAL '(+ 2 3)) to designate the *numeral* 5, since that numeral is the (local) procedural consequence of the application (+ 2 3). EVAL is extensional, as well. These lead to the following characterisation:

$\Phi E_0F_0("EVAL) = EXT(\Psi EF)$ (S3-60)

Unfortunately, however, this is ill-formed; the context arguments must be picked up explicitly. Thus we have:

$\Phi E_0F_0("EVAL) = \lambda E.\lambda F.\lambda S\ [\Psi EF(\Phi EF(CAR(S)))]$ (S3-61)

For example, suppose in some environment E_1 the variable X is bound to 100 and the variable Y to a pair P5, and in field F_2 that pair has CAR of 7. We then have:

$\Phi E_1F_2("(EVAL\ '(+\ X\ (CAR\ Y))))$ (S3-62)
$= [(\Phi E_1F_2("EVAL))E_1F_2]\ ["('(+\ X\ (CAR\ Y)))]$
$= [(\lambda E.\lambda F.\lambda S\ [\Psi EF(\Phi EF(CAR(S)))])E_1F_2]\ ["('(+\ X\ (CAR\ Y)))]$
$= [\lambda S\ [\Psi E_1F_2(\Phi E_1F_2(CAR(S)))]]\ ["('(+\ X\ (CAR\ Y)))]$

$= [\Psi E_1F_2(\Phi E_1F_2(CAR("'(+ X (CAR Y))))))]$
$= [\Psi E_1F_2(\Phi E_1F_2("'(+ X (CAR Y))))]$
$= \Psi E_1F_2("(+ X (CAR Y)))$

Since we have not yet spelled out Ψ, we are not yet in a position to continue this derivation, but the intent is clear. The correct context has been passed through, and what remains is merely to inquire as to the procedural consequence of the original argument in the context of use. Note that the original expression `(EVAL '(+ X (CAR Y)))` *designates* this result (namely, the value of the Ψ function of these arguments); that is also *evaluates* to this result will emerge only when we consider $\Psi EF(EVAL)$ in the next section.

The only other primitive we will consider is LAMBDA, and, rather than writing out the full meta-syntactic translation functions that construct an appropriate lambda calculus function designator from the arguments to the LAMBDA, we will instead simply describe in plain English what its declarative import comes to. The reason that we are beginning to ease up on mathematical rigour is that we already have plenty of ammunition to show how our present approach is doomed: after looking at LAMBDA we will show how, if we are to keep analysing 1-LISP, we will have to give up on *ever* using the extensionalisation function. Thus premature formalisation would be of no point.

As described in the last chapter, LAMBDA forms take a *type* argument to distinguish EXPRs from IMPRs. As we would expect, the declarative significance of expressions of the form `(LAMBDA EXPR <vars> <body>)` is that they designate functions, closed in the defining environment (this is 1.7-LISP), consisting of the lambda abstraction of `<vars>` over `<body>`. Such function designators are *extensional* — this is the crucial point. Thus, we will assume for the time being that we have a meta-linguistic translator function TRANS that takes four arguments: an environment and a field, and a variable list and a body (the first two meta-language objects, the second two syntactic objects of LISP), that designates the appropriate function. I.e. $TRANS(E_0,F_0,"(X),"(+ X 1))$ would designate the increment function (providing the atom + was bound as usual in E_0 to a designator of the extensionalisation of the addition function). Then in terms of this function the declarative import of LAMBDA can be described as follows:

$\Phi E_0F_0("LAMBDA)$ (S3-63)
$= \lambda E.\lambda F.\lambda S$ [*if* [S_1 = "*EXPR*]
then [EXT(TRANS(E,F,S_2,S_3))]
elseif [S_1 = "*IMPR*]
then [INT(TRANS(E,F,S_2,S_3))]]

where S = ⌜"($\underline{S_1}$, $\underline{S_2}$, $\underline{S_3}$)⌝

The crucial fact to notice about this characterisation is that the designation of *all* user-defined procedures are expressed in terms of EXT or INT. We have ourselves violated our original claim that we would always use one of these two; CAR, CDR, and LAMBDA have all had their own characterisations, because they needed explicit access to some aspect of the context of use above and beyond that provided by the extensionalisation and intensionalisation functions. What we have demonstrated, however, is that the exceptions to our convention are small in number and constrained: no others can be generated, because of this definition of LAMBDA.

That this characterisation is plausibly correct is manifested by two examples, one using the extensional and one using the intensional version. In particular, we will look at examples like those we used to show that a predicate EXT? was not definable. In that circumstance we had the following definitions:

```
(DEFINE F1 (LAMBDA EXPR (A) (CAR A)))                    (S3-64)
(DEFINE F2 (LAMBDA IMPR (A) (CAR A)))
```

and two examples of their use:

```
(F1 (CONS 3 4))      →     3                             (S3-65)
(F2 (CONS 3 4))      →     CONS
```

In order to avoid making use of DEFINE, which we have not yet analysed, and in order to avoid the CAR function, which needs explicit access to the field, we will instead consider the following two expressions:

```
((LAMBDA EXPR (A) A) (CONS 3 4))    →    (3 . 4)          (S3-66)
((LAMBDA IMPR (A) A) (CONS 3 4))    →    (CONS 3 4)
```

The semantical analysis is as follows. First we look at the designation of the two procedures:

ΦE_0F_0("*(LAMBDA EXPR (A) A)*) (S3-67)

= [(ΦE_0F_0("*LAMBDA*))E_0F_0] ["*(EXPR (A) A)*]

= [(λE.λF.λS [*if* S_1 = "*EXPR* ...])E_0F_0] ["*(EXPR (A) A)*]

= [λS [*if* S_1 = "*EXPR* ...]] ["*(EXPR (A) A)*]

= [*if* "*EXPR* = "*EXPR then* EXT(TRANS(E_0,F_0,"*(A)*,"*A*))
elseif S_1 = "*IMPR then* INT(TRANS(E_0,F_0,S_2,S_3))]

= EXT(TRANS(E_0,F_0,"*(A)*,"*A*))

= EXT(λX . X)

By an entirely similar proof we have as well:

$$\Phi E_0F_0(\text{"}(\textit{LAMBDA IMPR (A) A})) = \text{INT}(\lambda X\ .\ X) \qquad \text{(S3-68)}$$

Thus we can look at the two fuller applications:

$$\begin{aligned}&\Phi E_0F_0(\text{"}(\textit{(LAMBDA EXPR (A) A) (CONS 3 4)})) \qquad \text{(S3-69)}\\ &= [(\Phi E_0F_0(\text{"}(\textit{LAMBDA EXPR (A) A})))E_0F_0]\ [\text{"}(\textit{(CONS 3 4)})]\\ &= [(\text{EXT}(\lambda X\ .\ X))E_0F_0]\ [\text{"}(\textit{(CONS 3 4)})]\\ &= (\lambda X\ .\ X)\ [\Phi E_0F_0(\text{"}(\textit{CONS 3 4)})]\\ &= (\lambda X\ .\ X)\ [\text{"}(\textit{3 . 4)}]\\ &= \text{"}(\textit{3 . 4)}\end{aligned}$$

Analogously:

$$\begin{aligned}&\Phi E_0F_0(\text{"}(\textit{(LAMBDA IMPR (A) A) (CONS 3 4)})) \qquad \text{(S3-70)}\\ &= [(\Phi E_0F_0(\text{"}(\textit{LAMBDA IMPR (A) A})))E_0F_0]\ [\text{"}(\textit{(CONS 3 4)})]\\ &= [(\text{INT}(\lambda X\ .\ X))E_0F_0]\ [\text{"}(\textit{(CONS 3 4)})]\\ &= (\lambda X\ .\ X)\ [\text{"}(\textit{CONS 3 4)}]\\ &= \text{"}(\textit{CONS 3 4)}\end{aligned}$$

This is as much of an account, at least formulated in these simple terms, of the declarative semantics of LISP as we will examine for the present. We could go on: it would be possible to provide an fuller analysis of TRANS, for example, and we have not yet looked at APPLY (which would be the extensionalisation of a function of type $[[\ S \times S\ ] \rightarrow \ ^{\wedge}\ ]$ for 1-LISP and $[[\ \textit{FUNCTIONS} \times S\ ] \rightarrow D\ ]$ for 1.7-LISP). And we could look at lambda-binding of formal parameters, although the substantive question here has already been decided: we use environments as theoretical posits in the meta-language, and arrange for binding to preserve designation. However we have amassed ample evidence to be able to show much more serious problems with this approach than such incompleteness. One issue clearly has to do with side effects: we have modelled CAR and CDR, for example, but not CONS, because we have exhibited no mechanism by which the field can be affected; similarly, we have not examined SETQ or DEFINE, since the same point holds for the environment. Therefore we will turn, albeit briefly, to the procedural import of 1-LISP structures.

3.d.ii. Local Procedural Semantics (Ψ)

We turn next to the local procedural semantics (Ψ) of 1-LISP and 1.7-LISP: a characterisation of what, in those dialects' terminology, each type of s-expression evaluates to. ΨEF (i.e., Ψ relativised to context) is a function of type *[S → S]*; nonetheless, since we are still talking semantically, we are supposedly going to speak in terms of function application and so forth. An immediate and natural question is this: if both domain and range of Ψ are s-expressions, where will we find any functions to apply? Some of these s-expressions will *designate* functions, but that is of course of no help, because we have to characterise Ψ *independent* of the designation function Φ. Formulating a coherent reply to this concern will be the main emphasis in this brief sketch.

We could start to lay out Ψ mathematically, beginning with the obvious fact that in all contexts E,F, numerals return themselves:

$$\forall N \in \textit{NUMERALS},\ E \in \textit{ENV},\ F \in \textit{FIELDS}\ [\Psi EF(N) = N] \qquad \text{(S3-71)}$$

Before proceding in this fashion, however, we will instead look at a meta-circular interpreter, presented below (once again we concentrate on our "1.7-LISP" version of SCHEME, since it is more general than 1-LISP). This code for MC-EVAL is of interest for a variety of reasons. First, we can almost use this code directly to generate a mathematical account of Ψ, for the following reason:

> *It is the procedural consequence function that the meta-circular processor designates.*

Thus, at least approximately, we can almost assume that ΦEF("*MC-EVAL*) = EXT(ΨEF) (this fact will be crucial when we turn to the design of a reflective dialect). We as much as suggested this in the last section, albeit with reference to EVAL rather than to MC-EVAL. Of course in specifying that ΦEF("*EVAL*) = EXT(ΨEF) we were defining the semantics of EVAL, rather than defining Ψ. In the present instance, however, because we have defined MC-EVAL in terms of primitive procedures other than EVAL, the expression [ΦEF("*MC-EVAL*) = EXT(ΨEF)] (strictly, [ΦEF("*MC-EVAL*) = λE.λF.λS. [ΨEF(ΦEF(F¹(S)))]]) could in fact almost be used as a definition of Ψ.

A Meta-Circular 1.7-LISP Processor:

```
(DEFINE MC-EVAL                                                    (S3-72)
  (LAMBDA EXPR (EXP ENV)
    (COND ((MEMQ EXP '(T NIL)) EXP)
          ((NUMBERP EXP) EXP)
          ((ATOM EXP) (LOOKUP EXP ENV))
          (T (LET ((PROC (MC-EVAL (1ST EXP) ENV)))
                  (CASEQ (1ST PROC)
                     (P-IMPR (MC-APPLY-PI (2ND PROC) (REST EXP) ENV))
                     (P-EXPR (MC-APPLY-PE (2ND PROC)
                                          (MC-EVLIS (REST EXP) '() ENV)
                                          ENV))
                     (IMPR (MC-EVAL (4TH PROC)
                                    (BIND (3RD PROC) (REST EXP) (2ND PROC))))
                     (EXPR (MC-EVAL (4TH PROC)
                                    (BIND (3RD PROC)
                                          (MC-EVLIS (REST EXP) '() ENV)
                                          (2ND PROC)))))))))))

(DEFINE MC-APPLY-PI                                                (S3-73)
  (LAMBDA EXPR (FUN ARGS ENV)
    (CASEQ PROC
       (QUOTE  (1ST ARGS))
       (IF     (IF (NULL (MC-EVAL (1ST ARGS) ENV))
                   (MC-EVAL (3RD ARGS) ENV)
                   (MC-EVAL (2ND ARGS) ENV)))
       (LAMBDA (CONS (1ST ARGS) (CONS ENV (REST ARGS))))
       (DEFINE (SET-BIND (1ST ARGS) (MC-EVAL (2ND ARGS) ENV) ENV)))))

(DEFINE MC-APPLY-PE                                                (S3-74)
  (LAMBDA EXPR (FUN ARGS ENV)
    (CASEQ FUN
       (CAR     (CAR (1ST ARGS)))
       (CDR     (CDR (1ST ARGS)))
       (CONS    (CONS (1ST ARGS) (2ND ARGS)))
       (EQ      (EQ (1ST ARGS) (2ND ARGS)))
       (NUMBERP (NUMBERP (1ST ARGS)))
       (ATOM    (ATOM (1ST ARGS)))
       (READ    (READ))
       (PRINT   (PRINT (1ST ARGS)))
       (SET     (SET-BIND (1ST ARGS) (2ND ARGS) ENV))
       (+       (+ (1ST ARGS) (2ND ARGS)))
       (-       (- (1ST ARGS) (2ND ARGS)))
       (*       (* (1ST ARGS) (2ND ARGS)))
       (/       (/ (1ST ARGS) (2ND ARGS)))
       (EVAL    (MC-EVAL (1ST ARGS) ENV))
       (APPLY   (CASEQ (1ST (1ST ARGS))
                   (P-IMPR (ERROR 'YOU-CAN-ONLY-APPLY-EXPRS))
                   (IMPR   (ERROR 'YOU-CAN-ONLY-APPLY-EXPRS))
                   (P-EXPR (MC-APPLY-PE (2ND (1ST ARGS)) (2ND ARGS) ENV))
                   (EXPR   (MC-EVAL (4TH (1ST ARGS))
                                    (BIND (3RD (1ST ARGS))
                                          (2ND ARGS)
                                          (2ND (1ST ARGS))))))))))
```

```
(DEFINE MC-EVLIS                                                    (S3-76)
  (LAMBDA EXPR (ARGS ARGS* ENV)
    (IF (NULL ARGS)
        (REVERSE ARGS*)
        (MC-EVLIS (REST ARGS)
                  (CONS (MC-EVAL (1ST ARGS) ENV) ARGS*)
                  ENV))))

(MAPCAR (LAMBDA EXPR (FUN) (SET-FUNCTION FUN (LIST 'P-IMPR FUN)))   (S3-76)
       '(QUOTE IF LAMBDA DEFINE))

(MAPCAR (LAMBDA EXPR (FUN) (SET-FUNCTION FUN (LIST 'P-EXPR FUN)))   (S3-77)
       '(CAR CDR CONS EQ NUMBERP ATOM READ PRINT SET EVAL APPLY + - * /))
```

There are, however, a variety of reasons why we cannot adopt this suggestion literally. The first is relatively minor: it has to do with the fact that, as will be explained at the beginning of section 3.f, the present characterisation of Φ is wrong — it presumes that evaluation and interpretation can be identified, which we are of course at pains to show they cannot. In some cases our analysis is correct: for example, it would predict that in some context E,F the expression (MC-EVAL ''3) would *designate* the *local procedural consequence* of the *designation* of ''3. We know that ''3 designates the quoted expression '3, and we know that the expression '3 designates the numeral 3. Because '3 designates a numeral, and because a numeral is within the structural field, the evaluation theorem tells us that the local procedural consequence of '3 will be its referent: the numeral 3. Thus (MC-EVAL ''3) is supposed to designate that numeral. Hence, again, since numerals are part of the structural field, (MC-EVAL ''3) should *evaluate* to that numeral. We would correctly predict, in other words, the following:

```
(MC-EVAL ''3)    →    3                                             (S3-78)
```

Similarly, we have:

```
(MC-EVAL '3)     →    3                                             (S3-79)
```

This is predicted because '3 designates the numeral 3, and that numeral's procedural consequence is itself, and (MC-EVAL '3) should return that numeral. On the other hand, we also have:

```
(MC-EVAL 3)      →    3                                             (S3-80)
```

This, on our account, should generate an error, since the numeral 3 designates a *number*, and numbers do not have procedural consequences at all, not being expressions.

This kind of confusion will of course be repaired in 2-LISP. There is another reason that MC-EVAL is not the extensionalisation of Ψ, however, which is that whereas Ψ on our meta-linguistic account is a function of a *two-part* context — of an environment and a field — MC-EVAL takes only a single context argument: the environment. The reason is clear: the structural field is simply *there*, so to speak, accessible to examination and modification without further ado, because the meta-circular processor is *internal* to the computational process as a whole, whereas our meta-linguistic characterisation is of course entirely *external*. MC-EVAL still computes the field-relative procedural import; it obtains the field aspect of the context directly, however, without need of theoretically posited formal arguments.

This distinction between reified context arguments and directly accessible context fields will play a role in the characterisation not only of the full procedural consequence function Γ in the next section, but also in defining the 3-LISP processor in chapter 5.

There is another rather more serious reason why MC-EVAL does not quite represent what we are calling Ψ. In spite of being constructed in terms of procedures bearing the name "APPLY", MC-EVAL makes explicit the *formal* cut on procedural consequence: rather than *actually applying* the procedure (in an application) to the arguments, it performs the standard computational formal expression analogue of function application — a behaviour we will ultimately call *reduction*. It does not, therefore, clarify the question about closures and functions that we want to focus on.

Finally, MC-EVAL as just given is, as a declarative analysis of the code would make apparent, defined in terms of an environment *as a structure*, rather than as a function or list of pairs. In our mathematics we have defined Ψ as of type $[[\ ENVS \times FIELDS\ ] \rightarrow [\ S \rightarrow S\ ]]$; MC-EVAL is of type $[[\ S \times S\ ] \rightarrow S\ ]]$. We will make further comments on why it is reasonable to have Ψ defined in terms of abstract context, rather than in terms of structural context designators, in our review in section 3.f.iii; for the present we may simply observe that once again the use of an evaluative reduction scheme confuses use/mention issues almost irretrievably.

For all of these reasons, we will begin to erect our own characterisation of Ψ, therefore, by stepping through the definition of MC-EVAL line by line. As usual, we will begin with the numerals. Numerals evaluate to themselves in all environments:

$$\forall N \in \textit{NUMERALS},\ E \in \textit{ENV},\ F \in \textit{FIELDS}\ [\Psi EF(N) = N] \qquad \text{(S3-81)}$$

Similarly, the atoms T and NIL are self-evaluative; other atoms evaluate to their (procedural) bindings (not, of course, to what those bindings designate or return — we see here how the one notion of environment is used across both procedural and declarative significance):

$$\forall A \in \textit{ATOMS},\ E \in \textit{ENV},\ F \in \textit{FIELDS} \qquad \text{(S3-82)}$$
$$[\textit{if}\ [A \in \{\text{"}T,\ \text{"}NIL\}]\ \textit{then}\ [\Psi EF(A) = A]$$
$$\textit{else}\ [\Psi EF(A) = E(A)]]$$

These two equations mimic the first three COND clauses in S3-72 reproduced above.

The only other category are the pairs, encoding procedure applications. It is not immediately apparent how these should be treated: if we were to continue in a manner entirely parallel to our treatment of Φ, then we might expect something of the following sort for extensional procedures:

$$\forall S \in \textit{PAIRS},\ E \in \textit{ENV},\ F \in \textit{FIELDS} \qquad \text{(S3-83)}$$
$$[\Psi EF(S) = [\Psi EF(S_1)]\langle \Psi EF(S_2),\ \Psi EF(S_3),\ \ldots,\ \Psi EF(S_k)\rangle]$$
$$\textit{where}\ S = \ulcorner\text{"}(\underline{S_1}\ \underline{S_2}\ \underline{S_3}\ \ldots\ \underline{S_k})\urcorner$$

or, generalised to handle IMPRs as well as EXPRs (and assuming a definition of EXPR and IMPR as functions in the meta-language analogous to EXT and INT in the declarative case):

$$\forall S \in \textit{PAIRS},\ E \in \textit{ENV},\ F \in \textit{FIELDS} \qquad \text{(S3-84)}$$
$$[\Psi EF(S) = [(\Psi EF(S_1))EF]\ \langle S_2,\ S_3,\ \ldots,\ S_k\rangle]$$
$$\textit{where}\ S = \ulcorner\text{"}(\underline{S_1}\ \underline{S_2}\ \underline{S_3}\ \ldots\ \underline{S_k})\urcorner$$

and with such definitions of primitive procedures as this:

$$E_0F_0(\text{"}CAR) = EXPR(CAR) \qquad \text{(S3-85)}$$
$$E_0F_0(\text{"}+) = EXPR(+)$$
$$E_0F_0(\text{"}QUOTE) = IMPR(\lambda X.X)$$

The definition of EXPR would be the following:

$$EXPR \equiv \lambda G.\lambda E.\lambda F.\lambda S\ [G\langle \Psi EF(S_1),\ \Psi EF(S_2),\ \ldots,\ \Psi EF(S_k)\rangle] \qquad \text{(S3-86)}$$
$$\textit{where}\ S = \ulcorner\text{"}(\underline{S_1}\ \underline{S_2}\ \ldots\ \underline{S_k})\urcorner$$

The intuition behind these equations is this: just as extensional procedures de-referenced their arguments, so EXPRs should evaluate their arguments, and then apply their own "value" to those evaluated arguments.

There is however a serious problem with this approach, which brings to bring to light the fundamental problems that permeate these LISP dialects and the vocabulary

traditionally used to describe them. We said above that Ψ_{EF} was of type $[S \rightarrow S]$; EXPR, on the other hand, is a function that takes its arguments onto *functions*, which are not elements of the structural field. Thus S3-86 cannot be correct. It might seem that we could change E_0 to return an s-expression, but then equation S3-84 would have to fail, since s-expressions are not functions, and therefore cannot be used as such in the meta-linguistic characterisation.

We cannot, in other words, have the following two incompatible things: have Ψ take structures onto structures, and also have it take structures onto the functions that we attribute to them, *even the functions that represent their procedural import.* There are other problems of the same variety in the equations we just wrote down: S3-85 in conjunction with S3-84 would attempt to apply the real addition function (defined over numbers) to *numerals*, which represents a type-error in the meta-linguistic account. In sum, we will have to delineate a rational policy on use/mention issues before we can proceed with the definition of Ψ.

It is instructive to look at two places it might seem we could turn for help. Standard programming language semantics deals with functions, but they — as we made clear at the outset — deal with *designation*, and with *context modification*, not with structure-to-structure mappings of the program. Thus they would take "+" onto the addition function, which is not open to us. The meta-circular interpreter, of course, does remain with the structural domain, but it does not deal with functions. For primitive procedures like addition, it simply executes them in a non-inspectable fashion; for non-primitives, it would recursively decompose the structure encoding the definition. Thus for example if we were dealing with `(F 3)` where `F` had been defined in terms of `(LAMBDA (Y) (+ Y 1))`, the meta-circular processor would bind `Y` to the numeral `3` in an environment, and recursively process the expression `(+ Y 1)`. At some point in this process the primitive procedure `+` would be encountered, and the "addition" of the numeral `3` to the numeral `1` would be effected without explanation.

Thus neither of these two traditions affords any help. Note as well that there are two reasons we cannot appeal to the declarative interpretation function in order to turn the structure in procedure position into a function — cannot, that is, posit an equation of the following sort (where the underlined part is changed from S3-84):

$$\forall S \in \textit{PAIRS},\ E \in \textit{ENV},\ F \in \textit{FIELDS} \qquad \text{(S3-87)}$$
$$[\Psi EF(S) = [(\underline{\Phi E F}(S_1))EF]\ \langle S_2, S_3, \ldots, S_k\rangle]$$
$$\textit{where}\ S = \ulcorner(\underline{S_1}\ \underline{S_2}\ \underline{S_3}\ \ldots\ \underline{S_k})\urcorner$$

Not only is Φ not available to us in defining Ψ, but this would not even be correct. For + designates the real addition function, and $\Psi E(\ulcorner 1)$ is a *numeral*, not a number.

The only tenable solution — and, as mentioned in the introduction to this chapter, in fact a reasonable solution — is to define yet another interpretation function, from structures (since $\Psi EF(\ulcorner +)$ must be a structure) onto a *different function than its designation.* In the case of + the function we want is clearly what we may call the *numeral addition function*, defined as follows:

$$\lambda\langle A,B\rangle\ .\ M^{-1}(+(M(A),M(B)) \qquad \text{(S3-88)}$$

Such a function, in other words, given two numerals as arguments, yields that numeral that designates the sum of the integers designated by the two arguments.

That, of course, is exactly what one would expect the internal so-called "addition routines" to do. It is exactly what the "arithmetic" component of a CPU does. Furthermore, this is just the place where the idiosyncracies of representation would be taken care of. For example, in a particular version of LISP with fixed length integers (the LISPS we have defined, being abstract and infinite, do not have such limitations, and are therefore not quite physically realisable), the numeral addition function would not be described quite as simply as that given in S3-88 above, but rather shown to have limitations of one sort and another.

We will define a function, to be spelled "Δ", which maps a certain class of structures onto what we will call ***internal functions.*** We will call Δ the ***internaliser*** (to be distinguished from the *intensionalising* function of the preceding section). The internaliser is a function that takes closures, which are expressions, into functions from structures to structures; we will say that these functions are ***engendered*** by the closures. If we were to ignore its contextual relativisation, the internaliser would have the following type:

$$\Delta : [\,S \rightarrow [\,S \rightarrow S\,]] \qquad \text{(S3-90)}$$

In fact, however, contexts enter in; our initial version (we will have more complex versions subsequently) will take structures independent of context (since closures are context-independent) onto functions that are context-relative:

$$\Delta : [\ S \to [[\ ENVS \times FIELDS\ ] \to [\ S \to S\ ]]] \qquad \text{(S3-91)}$$

Then we have the following internal version of addition:

$$\Delta[E_0F_0(\text{"}+)] = \Delta[\text{"}(EXPR\ \underline{E_0}\ (A\ B)\ (+\ A\ B))] \equiv \lambda E.\lambda F\ .\ [\lambda\langle A,B\rangle\ .\ M^{-1}(+(M(\Psi EF(A)),M(\Psi EF(B))))] \qquad \text{(S3-92)}$$

Similarly, we will simply posit the value of Δ of all primitively recognised procedures. We will then enforce Δ to obey strict compositional rules for all non-primitive *EXPR* closures by defining it as follows. Note that the bound environment E is used to determine the significance of the arguments, but is not passed to the body S; instead, the formal parameters A_1 through A_k are bound on top of the *closure* environment E_c. This reflects the fact that 1.7-LISP is statically scoped.

$$\forall S \in S,\ A_1,A_2,\ \ldots A_k \in ATOMS,\ E_c \in ENVS \qquad \text{(S3-93)}$$
$$[\Delta\Gamma\text{"}(EXPR\ \underline{E_c}\ (\underline{A_1},\underline{A_2},\ \ldots \underline{A_k})\ \underline{S})\urcorner = \lambda E.\lambda F.\lambda\langle S_1,S_2,\ \ldots S_k\rangle\ \Psi E_1F(S)\ ]$$
$$\text{where } E_1 = E_c \text{ except that for } 1\leq i\leq k\ E_1(A_i)=\Psi EF(S_i).$$

We can then set out the following equation for the local procedural consequence of pairs:

$$\forall S \in PAIRS,\ E \in ENV,\ F \in FIELDS \qquad \text{(S3-94)}$$
$$[\Psi EF(S) = [(\Delta\Psi EF(S_1))EF]\ \langle S_2,\ S_3,\ \ldots\ ,\ S_k\rangle\ ]$$
$$\text{where } S = \Gamma\text{"}(\underline{S_1}\ \underline{S_2}\ \underline{S_3}\ \ldots\ \underline{S_k})\urcorner$$

As an example, consider $\Psi E_1F_1(\text{"}(+\ 2\ 3))$ (the atom + is assumed to have its standard binding in E_1):

$$\begin{aligned}
&\Psi E_1F_1(\text{"}(+\ 2\ 3)) \qquad \text{(S3-95)}\\
&= [\lambda E.\lambda F.[(\Delta\Psi EF(\text{"}+))EF]\ \langle\text{"}2,\ \text{"}3\rangle]\ E_1F_1\\
&= [(\Delta\Psi EF(\text{"}+))E_1F_1]\ \langle\text{"}2,\ \text{"}3\rangle\\
&= [(\lambda E.\lambda F\ .\ [\lambda\langle A,B\rangle\ .\ M^{-1}(+(M(\Psi EF(A)),M(\Psi EF(B))))])E_1F_1]\ \langle\text{"}2,\ \text{"}3\rangle\\
&= [\lambda\langle A,B\rangle\ .\ M^{-1}(+(M(\Psi E_1F_1(A)),M(\Psi E_1F_1(B))))]\ \langle\text{"}2,\ \text{"}3\rangle\\
&= M^{-1}(+(M(\Psi E_1F_1(\text{"}2)),M(\Psi E_1F_1(\text{"}3))))\\
&= M^{-1}(+(M(\text{"}2),M(\text{"}3)))\\
&= M^{-1}(+(2,3))\\
&= M^{-1}(5)\\
&= \text{"}5
\end{aligned}$$

As a second example we look at CAR. In 1-LISP's initial environment, that atom CAR is bound to a closure that engenders the actual CAR function:

$$\Delta\Psi EF(\text{"}CAR) = \Delta\text{"}(EXPR\ \underline{E_0}\ (A)\ (CAR\ A)) \equiv \lambda E.\lambda F\ .\ [\lambda\langle A\rangle\ .\ F^1(\Psi EF(A))] \qquad \text{(S3-96)}$$

To illustrate, consider $\Psi E_1F_0(\text{"}(CAR\ X))$ where X in E_1 is bound to the pair (HELLO . GOODBYE):

$$\begin{aligned}
&\Psi E_1F_0(\text{"}(CAR\ X)) \qquad (S3\text{-}97)\\
&\quad = [(\Delta\Psi EF(\text{"}CAR))E_1F_0]\ \langle\text{"}X\rangle\\
&\quad = [(\lambda E.\lambda F\ .\ [\lambda\langle A\rangle\ .\ F^1(\Psi EF(A))])E_1F_0]\ \langle\text{"}X\rangle\\
&\quad = [\lambda\langle A\rangle\ .\ F_0{}^1(\Psi E_1F_0(A))]\ \langle\text{"}X\rangle\\
&\quad = F_0{}^1(\Psi E_1F_0(\text{"}X))\\
&\quad = F_0{}^1(E_1(\text{"}X))\\
&\quad = F_0{}^1(\text{"}(HELLO\ .\ GOODBYE))\\
&\quad = \text{"}HELLO
\end{aligned}$$

This is course what it designates as well.

As a third and final example, we can look at QUOTE. Since QUOTE is primitive, its internal function has to be posited explicitly; we have:

$$\begin{aligned}
&\Delta\Psi EF(\text{"}QUOTE) = \Delta\text{"}(IMPR\ \underline{E_0}\ (A)\ A) \qquad (S3\text{-}98)\\
&\qquad\qquad \equiv \lambda E.\lambda F\ .\ \overline{[\lambda\langle A\rangle\ .\ A]}
\end{aligned}$$

Consider, for example, $\Psi E_1F_0(\text{"}(QUOTE\ X))$ for the same E_1 as in the previous example:

$$\begin{aligned}
&\Psi E_1F_0(\text{"}(QUOTE\ X)) \qquad (S3\text{-}99)\\
&\quad = [(\Delta\Psi EF(\text{"}QUOTE))E_1F_0]\ \langle\text{"}X\rangle\\
&\quad = [(\lambda E.\lambda F\ .\ [\lambda\langle A\rangle\ .\ A])E_1F_0]\ \langle\text{"}X\rangle\\
&\quad = [\lambda\langle A\rangle\ .\ A]\ \langle\text{"}X\rangle\\
&\quad = \text{"}X
\end{aligned}$$

Like the CAR example, we have shown that $\Psi EF(\text{"}(QUOTE\ X)) = \Phi EF(\text{"}(QUOTE\ X))$.

It is well to ask what is going on. In brief, what we are saying is that what we take the primitive procedures to designate has to be posited from the outside: this is what the lists of $E_0F_0(\langle\text{primitive-procedure}\rangle)$ were for. We have to posit *as well*, and *independently*, the functions that are computed by the primitive processing of those procedures. In specifying an applicative architecture, in other words, we have to do two things: we have to specify *how we are to interpret the functions*, and we have to specify *how the primitive functions are treated* (strictly, how procedure applications formed in terms of it are treated). Thus where we had the atom + designating the addition function, we also have now said that that atom engenders what we have called numeral addition, when processed by the primitive processor.

Given these two facts, we have just demonstrated a way in which the functions engendered by composite expressions can be determined from the functions engendered by the primitives. These functions — a class we are calling *internal* functions — are not the local procedural consequence of the primitive function designators, since by definition the local procedural consequence of any symbol must be a symbol. In addition, they are not what we take those primitive function designators to *designate*, because they cannot work

with abstract entities. Rather, they occupy a middle ground: they are presumably computed by the underlying implementation, and they additionally (one hopes) cohere in well-defined ways with the attributed functions they stand in an internal/external relationship to.

From this perspective, the internaliser "Δ" is neither odd nor awkward. In fact it brings to the fore a point about computation that underlies our entire account. We have assumed throughout that a computational device is a mechanism, the most natural explanation of which is formulated in terms of attributing semantics to its ingredients and operations. A computer, in other words, is a device whose behaviour is *semantically coherent.* Thus a pocket calculator or an abacus is computationally potent *under interpretation.* In spite of this, however, the behaviour is not itself the interpretation — to say that would involve a category error. These facts are exactly what our analysis makes plain: for primitive procedures, Φ tells us what our interpretation is; Δ tells us the function computed by the behaving mechanism.

In spite of this claimed naturalness, it is fortunate that in a rationalised dialect, once some global semantic properties can be proved, one rarely needs to traffic in these internal functions. If each of the primitives can be proved to cohere with the designated external functions, and if composition and so forth can be shown to work correctly, all predictions as to the consequence of structures can be mediated by the external attributed semantics. For us in our role as language, designers, however, these internal functions are for the meantime necessary.

This is as much of an exploration of local procedural consequence as we will take up, since it is limited to those procedures with no side effects. In order to handle more general circumstances, we will turn to the full consequence, described by the meta-linguistic function Γ.

3.d.iii. Full Procedural Consequence (Γ)

By the full procedural consequence we refer not only to what a given expression returns, but also to the full effect it has on both the structural field and the processor. We are modelling the field with a single theoretical posit; the processor by a pair of an environment and a continuation; thus our function Γ is of type $[[\ \mathbf{S} \times \mathbf{ENVS} \times \mathbf{FIELDS} \times$

CONTS $] \rightarrow [$ ***S*** $\times$ ***ENVS*** $\times$ ***FIELDS*** $]]$. That a continuation need not be part of the range of Γ is due to the way in which continuations work in an applicative setting, as the discussion in chapter 2 explained.

The meta-circular processor presented in the previous section (3.d.ii) dealt explicitly with environments as well as with structures; as we commented there, the field was not made an explicit argument, but was instead simply affected directly. Thus it was presumed that if `(MC-EVAL '(RPLACA X 'A))` was processed, and if `X` designated a structure accessible from outside, then that structure would be affected in a way in which the outside world would see. `MC-EVAL`, and the programs it processes, share the same field.

There is also a sense in which `MC-EVAL` and its processor share the same continuation structures. As the depth-first processing embodied in `MC-EVAL` causes levels of interpretation to nest, the partial result and so forth are maintained on the stack (an implementation of simple continuation strucutre) of the processor running `MC-EVAL`; no *explicit* continuation structure is maintained by `MC-EVAL` itself.

As we said in chapter 2, it is possible, using a higher-order dialect such as `1.7-LISP`, to model more explicitly the continuation structure involved in processing `LISP`. Thus we were led to what we called a "continuation-passing" meta-circular processor of the sort summarised below. As we can by now expect, this code is more similar to the characterisation of the full procedural consequence we are currently in search of.

A Tail-Recursive Continuation-Passing Meta-Circular 1.7-LISP Processor

```
(DEFINE MC-EVAL                                                    (S3-100)
  (LAMBDA EXPR (EXP ENV CONT)
    (COND ((MEMQ EXP '(T NIL)) (CONT EXP))
          ((NUMBERP EXP) (CONT EXP))
          ((ATOM EXP) (CONT (LOOKUP EXP ENV)))
          (T (MC-EVAL (1ST EXP) ENV
                (LAMBDA EXPR (PROC)
                   (CASEQ (1ST PROC)
                      (P-IMPR (MC-APPLY-PI (2ND PROC) (REST EXP) ENV CONT)
                      (P-EXPR (MC-EVLIS (REST EXP) '() ENV
                                  (LAMBDA EXPR (ARGS*)
                                     (MC-APPLY-PE (2ND PROC) ARGS* ENV CONT))))
                      (IMPR (MC-EVAL (4TH PROC)
                                     (BIND (3RD PROC) (REST EXP) (2ND PROC))
                                     CONT))
                      (EXPR (MC-EVLIS (REST EXP) '() ENV
                               (LAMBDA EXPR (ARGS*)
                                  (MC-EVAL (4TH PROC)
                                           (BIND (3RD PROC) ARGS* (2ND PROC))
                                           CONT)))))))))))

(DEFINE MC-EVLIS                                                   (S3-101)
  (LAMBDA EXPR (ARGS ARGS* ENV CONT)
    (IF (NULL ARGS)
        (CONT (REVERSE ARGS*))
        (MC-EVAL (CAR ARGS)
                 ENV
                 (LAMBDA EXPR (ARG*)
                    (MC-EVLIS (CDR ARGS) (CONS ARG* ARGS*) ENV CONT))))))

(DEFINE MC-APPLY-PI                                                (S3-102)
  (LAMBDA EXPR (PROC ARGS ENV CONT)
    (CASEQ PROC
       (QUOTE  (CONT (1ST ARGS)))
       (IF     (MC-EVAL (1ST ARGS) ENV
                   (LAMBDA EXPR (RESULT)
                      (IF (NULL RESULT)
                          (MC-EVAL (3RD ARGS) ENV CONT)
                          (MC-EVAL (2ND ARGS) ENV CONT)))))
       (LAMBDA (CONT (CONS (1ST ARGS) (CONS ENV (REST ARGS)))))
       (DEFINE (MC-EVAL (2ND ARGS) ENV
                   (LAMBDA EXPR (PROC)
                      (CONT (SET-BIND (1ST ARGS) PROC ENV))))))))

(DEFINE MC-APPLY-PE                                                (S3-103)
  (LAMBDA EXPR (PROC ARGS ENV CONT)
    (CASEQ PROC
       (EVAL    (MC-EVAL (1ST ARGS) ENV CONT))
       (APPLY   (CASEQ (1ST (1ST ARGS))
                   (P-IMPR (ERROR 'YOU-CAN-ONLY-APPLY-EXPRS))
                   (IMPR   (ERROR 'YOU-CAN-ONLY-APPLY-EXPRS))
                   (P-EXPR (MC-APPLY-PE (2ND (1ST ARGS)) (2ND ARGS) ENV CONT))
                   (EXPR   (MC-EVAL (4TH (1ST ARGS))
                                (BIND (3RD (1ST ARGS)) (2ND ARGS) (2ND (1ST ARGS)))
                                CONT))))))
```

```
(T (CONT (CASEQ PROC
            (CAR     (CAR (1ST ARGS)))
            (CDR     (CDR (1ST ARGS)))
            (CONS    (CONS (1ST ARGS) (2ND ARGS)))
            (EQ      (EQ (1ST ARGS) (2ND ARGS)))
            (NUMBERP (NUMBERP (1ST ARGS)))
            (ATOM    (ATOM (1ST ARGS)))
            (READ    (READ))
            (PRINT   (PRINT (1ST ARGS)))
            (+       (+ (1ST ARGS) (2ND ARGS)))
            (-       (- (1ST ARGS) (2ND ARGS)))
            (*       (* (1ST ARGS) (2ND ARGS)))
            (/       (/ (1ST ARGS) (2ND ARGS)))
            (SET     (SET-BIND (1ST ARGS) (2ND ARGS) ENV))))))))

(MAPCAR (LAMBDA EXPR (NAME) (SET-BIND NAME (LIST 'P-IMPR NAME) GLOBAL))    (S3-104)
      '(QUOTE IF LAMRDA DEFINE))

(MAPCAR (LAMBDA EXPR (NAME) (SET-BIND NAME (LIST 'P-EXPR NAME) GLOBAL))    (S3-105)
      '(CAR CDR CONS EQ NUMBERP ATOM READ PRINT SET EVAL APPLY + - * /))
```

As with the local case, we cannot simply take this to literally encode the full procedural consequence, for a number of reasons: the field is not explicitly mentioned, the environment is encoded as a structure, not as an abstract function, and 1-LISP's evaluation protocol wreaks its usual havoc. In 2-LISP it would be more possible to define the full consequence in terms of the full continuation-passing processor, but the field problem would remain. A solution to this, of course, is to pass the field as an explicit argument: this violates, however, the code's claim to being *meta-circular*; we would then be dealing with a full implementation of LISP in LISP. However the general claim that the *denotation* of an *implementation* of a computational process should be the *full procedural consequence* of the *implemented* language remains true.

It will turn out, however, as the next section will make plain, that Φ cannot ultimately be defined except in terms of Γ; thus we cannot define Γ by using Φ (although such a boot-strapping technique would be possible if a non-side-effect version of Γ were implemented, by using the Φ of 3.d.i, but we will not pursue such an approach). As mentioned in the introduction, we will not concentrate on Γ, but it is instructive to set out a few of its simple constraining equations.

The numerals are always straightforward:

$$\forall S \in \textit{NUMNERALS},\ E \in \textit{ENVS},\ F \in \textit{FIELDS},\ C \in \textit{CONTS} \qquad \text{(S3-108)}$$
$$[\Gamma(S,E,F,C) = C(S,E,F)]$$

Similarly the atoms:

$$\forall S \in \textit{ATOMS},\ E \in \textit{ENV},\ F \in \textit{FIELDS},\ C \in \textit{CONTS} \qquad \text{(S3-109)}$$
$$[\Gamma(S,E,F,C) = \textit{if } [S \in \{\text{"}T, \text{"}NIL\}] \textit{ then } [C(S,E,F)] \textit{ else } [C(E(S),E,F)]]$$

Of more interest is the characterisation of the full significance of pairs. In order to allow for side-effects, the idea is to allow the environment and field to percolate through the establishing of the significance of the constituent parts, so as to mirror the temporal flow of the processing:

$$\forall S \in \textit{PAIRS},\ E \in \textit{ENV},\ F \in \textit{FIELDS},\ C \in \textit{CONTS} \qquad \text{(S3-110)}$$
$$[\Gamma(S,E,F,C) \equiv [\Gamma(F^1(S),E,F,[\lambda\langle S_1,E_1,F_1\rangle\ .\ [\Delta S_1(F_1{}^2(S),E_1,F_1,C)]]]]$$

This version of Δ is a full context-passing version of the internaliser shown previously. Δ for addition, for example, is:

$$\Delta E_0F_0(\text{"}+) \qquad \text{(S3-111)}$$
$$\equiv \lambda S.\lambda E.\lambda F.\lambda C\ .\ [\Gamma(F^1(S),E,F,[\lambda\langle A,E_1,F_1\rangle\ .\ [\Gamma(F_1{}^1(F_1{}^2(S)),E_1,F_1,[\lambda\langle B,E_2,F_2\rangle\ .\ C([M^{-1}(+(M(A),M(B)))],E_2,F_2)]]]]$$

Similarly, the full internalisation of CAR is:

$$\Delta E_0F_0(\text{"}CAR) \qquad \text{(S3-112)}$$
$$\equiv \lambda S.\lambda E.\lambda F.\lambda C\ .\ [\Gamma(F^1(S),E,F,[\lambda\langle A,E_1,F_1\rangle\ .\ C([F^1(A)],E_1,F_1)]]$$

Finally, we posit the internalisation of QUOTE:

$$\Delta E_0F_0(\text{"}QUOTE) \equiv \lambda S.\lambda E.\lambda F.\lambda C\ .\ C(F^1(S),E,F) \qquad \text{(S3-113)}$$

As we did in the previous section, we can define the full internalisation Δ of composite (non-primitive) closures as follows:

$$\forall S \in \textit{S},\ A_1,A_2,\ldots A_k \in \textit{ATOMS},\ E \in \textit{ENVS} \qquad \text{(S3-114)}$$
$$[\Delta\Gamma\text{"}(\textit{EXPR } \underline{E}\ (\underline{A_1},\underline{A_2},\ldots \underline{A_k})\ \underline{S})\urcorner = \lambda S_0.\lambda E_0.\lambda F_0.\lambda C\ [\Gamma(F^1(S),E_0,F_0,[\lambda\langle V_1,E_1,F_1\rangle\ .\ [\Gamma(F_1{}^1(F_1{}^2(S)),E_1,F_1,[\lambda\langle V_2,E_2,F_2\rangle\ .\ \ldots\ [\lambda\langle V_k,E_k,F_k\rangle\ .\ \Gamma(S,E^*,F_k,[\lambda\langle S_c,E_c,F_c\rangle\ .\ C(S_c,E_k,F_c)])]\ldots]]]]]$$
$$\textit{where } E^* \textit{ is like } E \textit{ except that for } 1\leq i\leq k\ E^*(A_i)=V_i.$$

Each of the arguments, in other words, is processed in the environment of the call, with side effects passed from one to the next. When the body is finally processed, however, the environment given it is not the one which has sustained the processing of the arguments, but rather the closure environment extended to include bindings of the formal parameters to the new bindings. Note on return, however, that the environment passed to the continuation is ε_k (which may have been modified in the course of processing the arguments), not the (possibly modified) version of ε^* returned as a result of processing the body of the procedure. This arrangement is quite different from the case of the field, which is passed through the arguments to the body and thence directly to the continuation.

3.e. The Semantics of 1-LISP: a Second Attempt

It is time to take a step back from details for a spell, to reflect on what we have accomplished. On the face of it, we laid out a tentative declarative semantics for all of the 1-LISP structural types, and for all of its primitive procedures; similarly for both the local procedural semantics, and for the full procedural consequence. There would remain, of course, a tremendous amount of work before a complete semantics would be in place: the entire subject of functional composition, recursion, lambda abstraction (i.e., what TRANS comes to), variable binding, and so forth, would require treatment. Some of these subjects will arise in subsequent discussion of the dialects we build: the semantics of recursion, for example, will come into the foregoround when we discuss the 2-LISP implementation of recursive procedures in terms of an explicit Y-operator. However, as suggested earlier, we will not proceed with such considerations here, for a rather serious reason: our current approach is in trouble. There are a variety of problems that mean not only that our current results cannot be adopted intact, but more seriously that our *approach* cannot even be maintained. It will be instructive to show just how seriously what we have done so far is in error.

There are two sources of difficulty: one having to do with the semantical inelegance of evaluation, and one with temporal considerations and side effects. It is important to separate them, because the first set of problems are 1-LISP specific: in a rationalised dialect they could be corrected. The second, however, would confront any possible dialect of LISP; furthermore, they would appear to challenge the coherence of our maintaining that Φ and Ψ are distinct. Though we will show that this challenge can in fact be met — and our original intuitions preserved — to do so will lead us into some complexities.

3.e.i. The Pervasive Influence of Evaluation

The first concern is this: we have arranged it so that applications in terms of extensional procedures are defined with respect to the *designation* of the arguments — indeed, this is what it is to be an extensional procedure. However we have also assumed that all procedures defined as *EXPRs* are *extensional:* that procedures that procedurally are treated with EXPR can declaratively be treated with EXT. In 1-LISP, of course, this is not so.

Alternatively, to put the same point another way, if we assume this correspondence, we will never be able to describe how the procedural consequence and the declarative import of a given expression relate. The problem is that EXPRs *evaluate* their arguments, and evaluation, as we have time and again said, bears a strange relationship to designation. To show an example, we only have to show how, on the readings we have assumed, the expression (EQ 3 '3) designates falsity, but evaluates to T — presumably an unwelcome result.

The source of this particular problem was our too-hasty assumption that the primitive procedure EQ designates an extensional equality predicate. There is of course no doubt that from a procedural point of view it is an equality predicate: $\Psi E_0F_0(\text{"}EQ) = EXPR(=)$. Declaratively, however, we cannot get away with what seemed only natural: our claim that $\Phi E_0F_0(\text{"}EQ) = EXT(=)$. For consider the following:

$$
\begin{aligned}
&\Phi E_0F_0(\text{"}(EQ\ 3\ '3)) \qquad\qquad (S3\text{-}121)\\
&\quad = [(\Phi E_0F_0(\text{"}EQ))E_0F_0]\ (\text{"}(3\ '3))\\
&\quad = [(\Phi E_0F_0(E_0(\text{"}EQ))E_0F_0]\ (\text{"}(3\ '3))\\
&\quad = [(EXT(=))E_0F_0]\ (\text{"}(3\ '3))\\
&\quad = [((\lambda G.\lambda E.\lambda F.\lambda S\ G(\Phi E(S_1), \Phi E(S_2))) =)E_0F_0]\ (\text{"}(3\ '3))\\
&\quad = [(\lambda E.\lambda F.\lambda S\ =(\Phi EF(S_1), \Phi EF(S_2)))\ E_0F_0]\ (\text{"}(3\ '3))\\
&\quad = [\lambda S\ =(\Phi E_0F_0(S_1), \Phi E_0F_0(S_2))]\ (\text{"}(3\ '3))\\
&\quad = [=(\Phi E_0F_0(\text{"}3), \Phi E_0F_0(\text{"}'3))]\\
&\quad = [=(3, \text{"}3)]\\
&\quad = \text{False}
\end{aligned}
$$

This in spite of the fact that (EQ 3 '3) unarguably evaluates to T. The problem, of course, is that the expression '3 designates the *numeral* 3, whereas 3 designates the *number 3*. We have known this all along. Because of the evaluation theorem, however, these two sub-expressions *evaluate* to the same entity (the numeral). To make a proper definition of the designation of EQ, then, we would have to re-define it along roughly the following lines:

$$E_0F_0(\text{"}EQ) = INT(\lambda S_1,S_2\ .\ [\Psi EF(S_1) = \Psi EF(S_2)]) \qquad (S3\text{-}122)$$

except of course this (like all attempts to use INT when we want the meta-linguistic function itself to do some de-referencing or processing) is ill-formed — E and F aren't bound. Thus we are led to:

$$E_0F_0(\text{"}EQ) = \lambda E.\lambda F.\lambda S_1,S_2\ .\ [\Psi EF(S_1) = \Psi EF(S_2)] \qquad (S3\text{-}123)$$

EQ is just an example: we would have to recast *every* extensional procedure, making the function EXT of no use whatsoever. We would have to give up the intuition that *any* procedure was defined over the referents of its arguments, and recast them all as defined

over the *values* of their args. And this in order to establish the *designation* of the whole: we would claim that (EQ 3 '3) *designates truth* because 3 and '3 *evaluate* to the same numeral. We could generalise this approach, and define a meta-theoretic function EXPR that cast the designation in terms of the values of the arguments, but this would be absurd. For one thing, the designation of an expression would never be used: although you could use the meta-theoretic machinery to ask of a given expression what its designation was, the answer would be formulated in terms of the local procedural consequence of the ingredients, not in terms of the designation of anything! Looked at from the other side, we can see that from the fact that, for some expressions X and Y, the expression (EQ X Y) evaluates to T, one cannot say whether X and Y are co-designative — all one can say is that they are co-evaluative. So much the worse for 1-LISP.

The repair suggested in S3-123, in other words, attempts to solve the problem by dismissing it. It says that we have to abandon any notion of pre-theoretic attribution of semantics, in order to formulate an explicit account of that pre-theoretic attribution, which is nonsensical. To follow such an approach is to get lost in formalism and lose touch with our goals. It was our original aim to demonstrate the natural declarative attribution of significance to expressions formed in terms of EQ, which is undeniably that its arguments are the same. This last maneouver is an attempt to correct the declarative semantics so that the equations work out: a better strategy, we claim, is to correct LISP so that the natural intuitions are true of it.

3.e.ii. The Temporal Context of Designation

The second problem with the approach of the last section, in contrast, must squarely be faced. It is this: we have assumed, throughout our analysis, that the context in which an expression is used is always passed *down* through the tree being interpreted: thus, the environment in which each of the elements of a procedure application are interpreted is the same. In actual 1-LISP, however, the story is not so simple, because of side effects. Consider for example:

```
(LET ((A '(2 3)))                                          (S3-124)
    (+ 1 (BLOCK (RPLACA A 5)
                (CAR A))))
```

It is clear that this will *evaluate* to the numeral 6 (this would be reflected by looking at Γ); although we have not spelled out the declarative import of BLOCK, it should be evident that it will designate whatever is designated by the last form in its scope. By our discussion in section 3.c.v, where we admitted that the context of use of an expression, *for declarative as well as for procedural purposes*, was *temporally* as well as *structurally* located, we are forced to admit that (CAR A) in the form given must *designate* the number five; thus the whole must designate six. The equations we have set down, however, would not reflect the changed field in establishing the designation of (CAR A); thus they would predict that the *designation* of the whole was the number three.

It is for reasons like this, of course, that standard programming langauge semantics turned to continuation-passing style to encode the potential temporal interactions between the evaluation of one fragment of the code and another. We too took this approach, but only for *procedural* purposes. The present example would seem to suggest that we will have to do this as well for the declarative semantics, but such a suggestion looks, at first blush, as if it would violate our overall conception of procedural and declarative semantics as *distinct.*

This concern, however, is shallow. The answer is this: what differentiates *full* procedural consequence from *local* procedural consequence is that the former makes explicit all of the potential causal interactions between the processing of one part of a composite expression and another. The declarative semantics, by our own admission, is equally vulnerable to such causal effects. A full theory, therefore, *even of the declarative semantics*, should be, like Γ, formulated with full continuations, explicit field and environment arguments, and the rest. In other words, early in the chapter we argued that Ψ and Φ must be *separated*, but in the formal analysis that resulted we separated them too much; what we must now do is let them come back closer together, without losing grip on our claim that they describe different matters.

It is worth examining a variety of possible solutions to this problem of relating side-effects and other non-local procedural consequences with the declarative reading, for they illuminate several aspects of our approach. First, it would be possible to define the dialect simply without side effects. This is not quite as limiting as it might seem, given our overall interest in reflection. It is of course our long-range goal to define 3-LISP: in that dialect,

the ability to reflect is sufficiently powerful that one can obtain, in virtue of the very architecture, explicitly articulated meta-theoretic accounts of the procedural semantics of the underlying machine. It should be noted, as well, that side-effects and non-local control operators can obviously be *described* perfectly adequately in a language without side effects. Throughout our meta-theoretic analysis, for example, we have formulated Γ, which makes side-effects explicit, in the untyped λ-calculus — which is certainly a side-effect-free formalism. From these two points we can see how a reflective dialect of the pure non-side-effect λ-calculus would be sufficiently powerful so that procedures with "side-effects" (i.e., procedures behaviourally indistinguishable from those we say have side-effects in 1-LISP) could be defined. The strategy would be to define such procedures — SETQ and RPLACA and so forth — as reflective procedures that explicitly call the continuation with arguments designating the modified field and environment functions. For example a definition of SETQ might look something like the following. (To handle *field* side-effects would require passing the structural field as an explicit argument, which we do not do in 3-LISP, as discussed in section 5.a. Also, this code assumes an environment protocol like that shown later in S3-137; since in 3-LISP we in fact support environment side-effects primitively, environments are dealt with differently. But the following code *would* work if that scheme were adopted.)

```
(DEFINE SETQ                                                    (S3-125)
   (LAMBDA REFLECT [[VAR VAL] ENV CONT]
      (NORMALISE VAL ENV                              ; This is 3-LISP
         (LAMBDA SIMPLE [N-VAL]
            (CONT [N-VAL] (PREP [VAR N-VAL] ENV))))))
```

The syntax and meaning of this will of course be explained only in chapter 5, but the intent is this: a call to SETQ (say, (SETQ X 4)) would reflect upwards one level, binding VAR and VAL to designators of X and 4, and binding ENV and CONT to the environment and continuation in effect at the time of the call. After normalising the value (the variable doesn't need to be normalised, because this is SETQ, the continuation is called with the normalised value not only as the result, but with an environment in which the binding of the variable to the normalised "value" tacked on the front.

RPLACA and CONS and so forth could be similarly constructed. However to do this would be an empty victory, for all that would have happened would be that the semantical account of side-effects would be buried inside the definition of SETQ, rather than made

explicit in the semantical account of SETQ; *the full significance of* SETQ *would be identical either way.* From the fact that side-effects can be *described* nothing of particular note follows; our problem was how they were to interact with designation. If we adopted the approach just given we would have to say that the designation of any expression depends on the designation of any reflective procedures in its arguments, which is merely a recasting of the same problem — a recasting, it should be noted, into a *much* more difficult subject matter. At the present time the author has no suggestions as to how the semantics of 3-LISP can be finitely described (although there seems no doubt that they *could* be — we merely lack techniques). This is one reason that 2-LISP merits development on its own, where semantical characterisation is still tractable.

A second possible approach to the problem of procedural dependencies would be to give up, when faced with side-effects: to say, in a case where the arguments to a procedure involve side-effects, that we have no principled way of saying what the designation of the whole form is. We would simply decline to specify the designation of (BLOCK (SETQ A 3) 3), for example. This, however, is an admission of defeat — and, we will be at pains to argue, an *unnecessary* defeat. What it amounts to is a claim that the temporal aspects of the context of use of an expression not only *affect* the designation of that expression, but that they affect it in ways which we cannot describe. But of course we *can* describe the temporal aspects of the context — the full procedural semantics function Γ was developed exactly in order to make them explicit. Therefore it seems *unlikely* that we cannot describe their declarative effect. Thus this second option should also be rejected — particularly because it is not so much an option as a suggestion that we abandon the effort.

The only approach still open, then, is this: we should allow that procedural consequence can affect designation, and try to lay out the ways in which Φ will depend on the contextual modification made explicit by Γ. At first blush this would seem to connect the declarative and procedural notions so closely that we lose the ability to prove the evaluation theorem, for the whole argument at the beginning of the chapter focused on how it was essential to have declarative and procedural readings specified *independently* in order to *prove* anything about how they relate. However we will not, as it happens, be in such deep water as all that, as the next pages should make clear.

3.e.iii. Full Computational Significance (Σ)

The approach is simply to identify very carefully our assumptions — including the admission that the declarative import of a symbol may be a function of its temporal as well as its structural context — and proceed once again to erect the mathematical machinery to honour them. The examples in the last few paragraphs have indicated that only the *full* computational account of the processing of symbols will enable us to determine the declarative import of an expression. Does this mean that that full computational account *is* the declarative import? Of course not. Does it mean that the local procedural consequence and the declarative import merge? No, there is no need for that. It is helpful to remember that the original intuition in the case of numerals — that numerals designate numbers but return themselves — is simple and perfectly coherent. No matter how complex other circumstances force our analysis to be, we should never feel the need to give up the ground cases.

One possibility would be to formulate a *full declarative semantical function* — called Π, say — that would stand in the same relationship to Φ that Γ stands to Ψ. However this is wrong-headed: as we mentioned earlier, although the declarative import of an expression is *affected* by procedural consequence, it does not itself *affect* context. Thus the situation is not symmetric, and defining such a Π would *duplicate* much of Γ. What we want, instead, is to show how Φ *depends* on the contextual modifications that are already adequately manifested by Γ.

The approach we will follow is to adopt a new, fully general, computational semantical function — a kind of "grand interpreter" — which is formulated not purely in aid of the local procedural consequence, but which instead makes clear how that procedural consequence affects the full context of each expression, for both declarative and procedural purposes. The natural suggestion is to have the new semantical interpretation function convey *both* the procedural and declarative import, as well as the context information. Thus, whereas Γ mapped structures and contexts onto structures and contexts, we will examine a function that maps structures and contexts onto *structures, designations, and contexts.* More precisely, we will have a new ***full computational significance*** function (which we will call Σ for alliterative reasons), of type:

$$[[\ S \times ENVS \times FIELDS \times CONTS\ ] \rightarrow [\ S \times D \times ENVS \times FIELDS\ ]] \qquad \text{(S3-126)}$$

Thus in a given context we will say that a computational expression ***signifies*** a four-tuple of a result, a designation, and a two-part resultant context.

The intent, in a case where there are no side effects, will be roughly the following (this is ill-defined, but is intended to convey the overall flavour):

$$\forall S,\ E,\ F,\ C\ [\Sigma(S,E,F,C) = C(\Psi EF(S),\ \Phi EF(S),\ E,\ F)] \qquad \text{(S3-127)}$$

It is to be noted, however, that we will *define* Ψ and Φ in terms of Σ, so the above equation is for the moment strictly content-free.

In order to convey a sense of this new Σ, we can characterise in its terms the corresponding new formulation of the evaluation theorem:

$$\begin{aligned}&\forall S_1,\ S_2 \in S,\ E_1,\ E_2 \in ENV,\ F_1,\ F_2 \in ENV,\ D \in D \qquad \text{(S3-128)}\\ &\quad [[\ \Sigma(S_1,E_1,F_1,ID) = \langle S_2,\ D,\ E_2,\ F_2\rangle\ ] \supset\\ &\quad [\ \mathit{if}\ [D \in S]\ \mathit{then}\ [S_2 = D]\ \mathit{else}\ [\Phi E_1F_1(S_2) = D]\ ]]\end{aligned}$$

Similarly, we would have a new statement of the normalisation theorem:

$$\begin{aligned}&\forall S_1,\ S_2 \in S,\ E_1,\ E_2 \in ENV,\ F_1,\ F_2 \in ENV,\ D \in D \qquad \text{(S3-129)}\\ &\quad [[\ \Sigma(S_1,E_1,F_1,ID) = \langle S_2,\ D,\ E_2,\ F_2\rangle\ ] \supset\\ &\quad [[\Phi E_1F_1(S_2) = D] \wedge [\text{NORMAL-FORM}(S_2)\ ]]]\end{aligned}$$

Note that in both cases the relationship betwen S_2 and D is expressed using Φ; we still need to discharge this reference (although this use of Φ is relativised to E_1F_1, since S_2 is in normal-form, E_2F_2 or any other context would serve as well). What we will do is to *define* Φ and Ψ in terms of the new Σ, so that they can be used as they were before. In particular, we make them selectors on the sequence of entities returned by Σ:

$$\begin{aligned}\Psi &\equiv \lambda E.\lambda F.\lambda S\,[\Sigma(S,E,F,\lambda X.[X^1])] \qquad \text{(S3-130)}\\ \Phi &\equiv \lambda E.\lambda F.\lambda S\,[\Sigma(S,E,F,\lambda X.[X^2])]\end{aligned}$$

Thus if we inquire as to ΦEF of a given expression Y, we are by these definitions taken to be asking about the first coordinate of the four-tuple designated by ΣEF of Y, given an essentially empty continuation. These not only make equations S3-128 and S3-129 meaningful; they enable us to shorten those formulations as well. In particular, we get the following restatements of our main theorems (evaluation and normalisation, respectively):

$$\begin{aligned}&\forall S \in S,\ E \in ENVS,\ F \in FIELDS \qquad \text{(S3-131)}\\ &\quad [\ \mathit{if}\ [\Phi EF(S) \in S]\\ &\qquad \mathit{then}\ [\Phi EF(S) = \Psi EF(S)\ ]\end{aligned}$$

else [[ΦEF(S) = ΦEF(ΨEF(S))] ∧ [NORMAL-FORM(ΦEF(S))]]]

∀S ∈ ***S***, E ∈ ***ENVS***, F ∈ ***FIELDS*** (S3-132)
[[ΦEF(S) = ΦEF(ΨEF(S))] ∧ [NORMAL-FORM(ΦEF(S))]]

Except for the increased complexity for dealing with environments and fields, these equations closely resemble the initial versions we presented in the chapter's introduction.

The use of the identity continuation in S3-130 is intentional, and deserves some comment. Σ must be formulated in terms of continuations, in order properly to handle side-effects of all kinds. Thus the full procedural consequence of a form such as (RETURN 10) is describable only in such terms. Note, however, what would happen if we asked what (RETURN 10) *designated*, or what (RETURN 10) *resulted in.* By S3-130, we would inquire as to the first or second element of the four-tuple signified by (RETURN 10) — i.e., designated by Σ("*(RETURN 10)*,E,F,λX.X). In all liklihood this would be ill-formed, since λX.X is not a continuation structured in the way that (RETURN 10) would require. But this is perfectly reasonable: (RETURN 10) does not really *have* a designation on its own. If, on the other hand, we ask for the designation or local procedural consequence of:

```
(PROG (I)                                                    (S3-133)
      (SETQ I 0)
    A (SETQ I (+ I 1))
      (IF (= I 4) (RETURN I))
      (GO A))
```

Then the answer will be the number four (or the numeral 4), and this will have been determined in virtue of examining the *full* computational significance of the embedded term (RETURN I), rather than examining only that term's local import. This careful trading between full signficance and local designation is just what we want: it is too broad to say that the designation *is* the full significance (that is what standard programming language semantics approximately does), but it is too narrow to say that the designation is formed only of the designation of the consituents (that was the error of the previous section). In this new formulation we retain the ability to talk about the local aspects — designation and result — of the full significance, but can still compose those local aspects out of the full significance of the ingredients. This is the point towards which we have been working this long while.

The easiest way to see what this reformulation amounts to is to begin laying out the characterisation, under this new protocol, of the semantics of the basic structural types and

the primitive procedures. The first three structural types are straightforward:

```
Σ ≡ λS.λE.λF.λC                                                    (S3-134)
    [case TYPE(S)
      NUMERAL → C(S, N(S), E, F)
      ATOM    → if [S ∈ {"T, "NIL}] then C(S, T(S), E, F)
                                    else C(E(S), ΦEF(E(S)), E, F)]
```

First we consider the numerals and booleans. In both cases what is *returned* is the numeral or boolean; what is *designated* is the integer or truth-value associated with the constant symbol. Neither environment nor field are affected, and the provided continuation is applied; thus both are context-independent and side-effect free. All is straightforward.

Atoms too are side-effect free, but they of course depend crucially on the environment. What is *returned* is the binding; what is *designated* is the designation, *in the context of use*, of that binding. It is, as we have noted before, only the fact that *bindings* are context-independent that legitimises this ostensibly odd characterisation of their designation.

The use of the continuation in S3-134 should be noted. It would seem that the full computational significance of a numeral should, rather than C(S, N(S), E, F), be <S, N(S), E, F>. C, after all, might be some continuation mapping that result onto some other unknown entity. However to ask what the computational significance of an expression is, we have to do so in a context. If we ask only with respect to a field and an environment, we use the identity function ID (ID is in this case $[\lambda\langle X_1,X_2,X_3,X_4\rangle . \langle X_1,X_2,X_3,X_4\rangle]$, since continuations are applied to four-tuples); thus, in some E_i and F_k, the full computational significance of the numeral 3 is Σ("3, E_i, F_k, ID), which *is* the sequence <S, N(S), E, F>. In a more complex case, however, we might ask for the designation of an expression that involves a control side-effect; in an appropriate context, the form (THROW 'TOP-LEVEL 3) might designate the number three; this could not be determined if continuation-passing semantics were not employed.

More revealing than the three atomic types is the full significance of pairs:

```
∀S ∈ PAIRS                                                         (S3-135)
  Σ(S) = λE.λF.λC
         [Σ(F¹(S), E, F,
             [λ⟨S₁,D₁,E₁,F₁⟩ .
                   [ΔS₁(F₁²(S),E₁,F₁,
                           [λ⟨S₂,E₂,F₂⟩ . C(S₂,D₁(F₁²(S),E₁,F₁),E₂,F₂)])]]]
```

In order to understand this, consider the various ingredients we have to deal with. First, in any pair the CAR — $F^1(S)$ in the equation — will presumably *designate* a function. Since the term $\Sigma(F^1(S), E, F, [\lambda\langle S_1, D_1, E_1, F_1\rangle . \ldots])$ will designate its fourth argument applied to the full significance of that CAR, we can presume (assuming that the CAR engenders no control side-effects) that D_1 will designate that function, and S_1 will designate the expression to which the CAR evaluates (a closure of some sort, presumably). For example, if S was the expression (+ 1 2), D_1 will designate (the extensionalisation of) the addition function, and S_1 will designate the +-closure (EXPR E0 (A B) (+ A B)).

There are then, as we have pointed out before, *three* ways in which we expect to combine these various ingredients (those ingredients being the closure, the addition function, and the arguments). Under one, the extensionalised function will be applied to the arguments: this is $D_1(F_1^2(S), E_1, F_1)$ (since D_1 is the *extensionalisation* of the addition function, this will apply the real addition function to the *designations* of the arguments, as expected). Under the second, the *internalised* version of that function will be applied to the arguments: this is $\Delta S_1(F_1^2(S), E_1, F_1, C)$. Under the third (the *formal* account), the closure will be reduced with the arguments by a computational process — this is what the meta-circular processor makes clear, but is not something that we try to manifest in the semantics.

The first two are represented in this code by making the internalised function take as an explicit continuation a function that receives the full *procedural* consequence of the application of that internalised function, but then puts this together with the *declarative* import of the application, which is calculuated independently in the meta-language. It should be evident that this technique, which essentially branches the meta-linguistic characterisation of the significance of pairs, calculating the procedural and declarative import separately, bears the brunt of the claim that the two are to be independently specified. From a *computational* point of view this meta-linguistic characterisation is inefficient, because both $D_1(F_1^2(S), E_1, F_1)$ and $\Delta S_1(F_1^2(S), E_1, F_1, C)$ recursively decompose in terms of Σ of their consituents, but it is exactly the difference between them that captures our foundational intuition that declarative and procedural import, although both dependent on the same computational contextualisation, are nonetheless distinct.

In order to illustrate the use of this reconstituted Σ, we will show the significance of several primitives. First we take CAR:

$$\Sigma(E_0F_0(\text{"}CAR)) = \lambda E.\lambda F.\lambda C\ .\ [C(\text{"}(EXPR\ \underline{E0}\ (X)\ (CAR\ X)),\ [\lambda\langle A,E_1,F_1\rangle\ .\ \Sigma(F_1^1(A),E_1,F_1,[\lambda\langle S_2,D_2,E_2,F_2\rangle\ .\ F_2^1(D_2)]])]\ E,\ F) \qquad \text{(S3-136)}$$

This can be described in English as follows. First, CAR is a procedure whose signficance is that, in a context (E and F), it straightforwardly signifies (calls C) with a tuple of four things, as usual: its normal-form, its designation, and the context unchanged (E and F, again). Thus right away we see that in this environment CAR is side-effect free. The normal-form (the s-expression "*(EXPR E0 (X) (CAR X))*) is the closure of the CAR procedure, which is described separately, below. The function that it designates is the crucial thing to look at. It is a function of three things (all designated functions are called with three arguments: an argument to the application and a two-part context — this is unchanged from before); it first obtains the significance of its single argument $F_1^1(A)$, in that context (E_1 and F_1). Then the crucial part comes: $F_2^1(D_2)$, which is of course the CAR of S_2 in field F_2. Thus applications in terms of CAR *designate* the first element of the pair designated by their arguments. This is entirely to be expected.

There are various things to be noted. First, that the CAR-closure *(EXPR E0 (X) (CAR X))* *designates* the second argument would have to be *proved* — this presumably can be done. Second, it is only the *designation* D_2 of the full significance of A_1 that is given to the CAR function (F_2^1). Otherwise the context returned by as E_2, F_2 is ignored. The full Σ-characterisation of CAR will pick those up explicitly, so there is no harm in ignoring them here.

The full internaliser Δ has to be mildly redefined so as to deal with a Σ that yields four-tuples, although it consistently ignores the denotations. We give first its new general definition on non-primitive closures. Note that C^* is not a full continuation, in the sense that it is a function of *three* arguments, not four (an example of such a C^* appeared in S3-136):

$$\forall S \in \mathbf{S},\ A_1,A_2,\ldots A_k \in \mathbf{ATOMS},\ E \in \mathbf{ENVS} \qquad \text{(S3-137)}$$
$$[\Delta\Gamma\text{"}(EXPR\ \underline{E}\ (\underline{A_1},\underline{A_2},\ldots\underline{A_k})\ \underline{S})\urcorner = \lambda S_0.\lambda E_0.\lambda F_0.\lambda C^* [\Sigma(F^1(S),E_0,F_0,\ [\lambda\langle V_1,D_1,E_1,F_1\rangle\ .\ [\Sigma(F_1^1(F_1^2(S)),E_1,F_1,\ [\lambda\langle V_2,D_2,E_2,F_2\rangle\ .\ \ldots\ [\lambda\langle V_k,D_k,E_k,F_k\rangle\ .$$

$$\Sigma(S,E^*,F_k,[\lambda\langle S_c,D_c,E_c,F_c\rangle\ .\ C^*(S_c,E_k,F_c)])]\ldots]]]]]$$

where E^* *is like* E *except that for* $1\leq i\leq k$ $E^*(A_i)=V_i$.

As before, we have to provide the internalisation of all primitives; for illustration we first present it for CAR:

$$\Delta E_0F_0(\text{"}CAR) \equiv \lambda S.\lambda E.\lambda F.\lambda C\ .\ [\Sigma(F^1(S),E,F,\ [\lambda\langle A,D_1,E_1,F_1\rangle\ .\ C([F_1^1(A)],E_1,F_1)]] \tag{S3-138}$$

We also give the full significance, and the internalisation, of + and QUOTE:

$$\Sigma(E_0F_0(\text{"}+)) = \lambda E.\lambda F.\lambda C\ .\ [C(\text{"}(EXPR\ \underline{E0}\ (B\ C)\ (+\ B\ C)),\ [\lambda\langle A,E_1,F_1\rangle\ .\ \Sigma(F_1^1(A),E_1,F_1,\ [\lambda\langle A_2,D_2,E_2,F_2\rangle\ .\ \Sigma(F_2^1(F_1^2(A)),E_2,F_2,\ [\lambda\langle A_3,D_3,E_3,F_3\rangle\ .\ +(D_2,D_3)])])]\ E,\ F) \tag{S3-139}$$

$$\Delta E_0F_0(\text{"}+) \equiv \lambda S.\lambda E.\lambda F.\lambda C\ .\ [\Sigma(F^1(S),E,F,\ [\lambda\langle A,D_1,E_1,F_1\rangle\ .\ [\Sigma(F_1^1(F^2(S)),E_1,F_1,\ [\lambda\langle B,D_2,E_2,F_2\rangle\ .\ C([M^{-1}(+(M(A),M(B)))],E_2,F_2)]]]] \tag{S3-140}$$

$$\Sigma(E_0F_0(\text{"}QUOTE)) = \lambda E.\lambda F.\lambda C\ .\ [C(\text{"}(IMPR\ \underline{E0}\ (X)\ X),\ [\lambda\langle A,E_1,F_1\rangle\ .\ F_1^1(A)]\ E,\ F) \tag{S3-141}$$

$$\Delta E_0F_0(\text{"}QUOTE) \equiv \lambda S.\lambda E.\lambda F.\lambda C\ .\ C(F^1(S),E,F) \tag{S3-142}$$

3.e.iv. An Example

To step through a particular example will be instructive (if a little tedious). We will look at the full significance of `(CAR '(A B C))` in 1-LISP under this new approach:

```
Σ("(CAR '(A B C)), E0, F0, ID)                                        (S3-143)
    = [Σ(F0¹("(CAR '(A B C))),
        E0,
        F0,
        [λ<S1,D1,E1,F1> .
            [ΔS1(F1²("(CAR '(A B C))),E1,F1,
                [λ<S2,E2,F2> .
                    (ID)<S2,D1(F1²("(CAR '(A B C))),E1,F1),E2,F2>])]]]
```

This is merely equation S3-135 with our particular arguments filled in — this is legitimate because `(CAR '(A B C))` is a pair. First we perform the CAR operations out of F_0:

```
    = [Σ("CAR, E0, F0,                                                (S3-144)
        [λ<S1,D1,E1,F1> .
            [ΔS1(F1²("(CAR '(A B C))),E1,F1,
                [λ<S2,E2,F2> .
                    (ID)<S2,D1(F1²("(CAR '(A B C))),E1,F1),E2,F2>])]]]
```

Now equation S3-136 applies, since E_0F_0("CAR) is primitive:

```
    = ([λ<S1,D1,E1,F1> .                                              (S3-145)
            [ΔS1(F1²("(CAR '(A B C))),E1,F1,
                [λ<S2,E2,F2> .
                    (ID)<S2,D1(F1²("(CAR '(A B C))),E1,F1),E2,F2>])]]
       <"(EXPR E0 (X) (CAR X)),
        [λ<A,E1,F1> . Σ(F1¹(A),E1,F1,[λS2,D2,E2,F2 . F2¹(D2)])]
        E0, F0>)
```

We perform the first reduction, allowing the significance of CAR to bind in a context and an argument structure. D_1 will bind to the extensionalisation of the CAR function; S_1 will bind to the closure that the *internaliser* will subsequently *also* take onto the CAR function, as we will see (thus preparing the way for the declarative and procedural consequence being the same).

```
    = ([Δ("(EXPR E0 (X) (CAR X)))]                                    (S3-146)
       <F0²("(CAR '(A B C))),
        E0,
        F0,
        [λ<S2,E2,F2> .
            (ID)<S2,
                ([λ<A,E1,F1> . Σ(F1¹(A),E1,F1,[λS2,D2,E2,F2 . F2¹(D2)])]
                 <F0²("(CAR '(A B C))),E0,F0>),
                E2,
                F2>]>)
```

We can remove the identity function, and perform the two CDRS on F_0:

```
= ([Δ("(EXPR E0 (X) (CAR X)))]                                    (S3-147)
   <"('(A B C)),
    E0,
    F0,
    [λ<S2,E2,F2> .
        <S2,
         ([λ<A,E1,F1> . Σ(F1¹(A),E1,F1,[λS2,D2,E2,F2 . F2¹(D2)])]
          <"('(A B C)),E0,F0>),
         E2,F2>]>)
```

Then we can proceed to calculate the declarative import. Note that we have not yet expanded the term representing the internalisation of the CAR closure; we are instead in the midst of calculating, from the semantical characterisation of the ingredients, what the whole expression *designates*. We will turn to the calculation of what it *returns* presently.

```
= ([Δ("(EXPR E0 (X) (CAR X)))]                                    (S3-148)
   <"('(A B C)),E0,F0,
    [λ<S2,E2,F2> .
        <S2,
         [Σ(F1¹("('(A B C))),E0,F0,[λS2,D2,E2,F2 . F2¹(D2)])],
         E2,F2>]>)
```

Once again performing a CAR off F_1 (and expanding the "'(A B C) to its full representation as "(QUOTE (A B C))):

```
= ([Δ("(EXPR E0 (X) (CAR X)))]                                    (S3-149)
   <"('(A B C)),E0,F0,
    [λ<S2,E2,F2> .
        <S2,
         [Σ("(QUOTE (A B C)),E0,F0,[λ<S2,D2,E2,F2> . F2¹(D2)])],
         E2,F2>]>)
```

This subsidiary call to Σ by the declarative significance of CAR is necessary in case there are side effects, of course, which our example will not illustrate. Nonetheless it affords a good example of the full significance of the paradigmatic IMPR. First we apply Σ in the general case for pairs (the internal continuation from above has been renamed, in an α-conversion, to use "3" subscripts, rather than "2", to avoid confusion):

```
= ([Δ("(EXPR E0 (X) (CAR X)))]                                    (S3-150)
   <"('(A B C)),E0,F0,
    [λ<S2,E2,F2> .
        <S2,
         [Σ(F0¹("(QUOTE (A B C))),
            E0,
            F0,
            [λ<S1,D1,E1,F1> .
                  [ΔS1(F1²("(QUOTE (A B C))),
```

```
                                E1,
                                F1,
                                [λ<S2,E2,F2> .
                                     ([λ<S3,D3,E3,F3> . F3¹(D3)]
                                      <S2,
                                       [D1(F1²("(QUOTE (A B C))),E1,F1)],
                                       E2,F2>)])])],
                E2,F2>])
```

Doing the CAR on F_0 extracts the QUOTE function explicitly:

```
= ([Δ("(EXPR E0 (X) (CAR X)))]                                          (S3-161)
   <"('(A B C)),E0,F0,
    [λ<S2,E2,F2> .
        <S2,
        [Σ("QUOTE,E0,F0,
           [λ<S1,D1,E1,F1> .
               [ΔS1(F1²("(QUOTE (A B C))),E1,F1,
                    [λ<S2,E2,F2> .
                         ([λ<S3,D3,E3,F3> . F3¹(D3)]
                          <S2,
                           [D1(F1²("(QUOTE (A B C))),E1,F1)],
                           E2,F2>)])])],
        E2,F2>])
```

Now equation S3-141 defining the procedural consequence of QUOTE applies:

```
= ([Δ("(EXPR E0 (X) (CAR X)))]                                          (S3-162)
   <"('(A B C)),E0,F0,
    [λ<S2,E2,F2> .
        <S2,
        ([λ<S1,D1,E1,F1> .
             [ΔS1(F1²("(QUOTE (A B C))),E1,F1,
                  [λ<S2,E2,F2> .
                       ([λ<S3,D3,E3,F3> . F3¹(D3)]
                        <S2,
                         [D1(F1²("(QUOTE (A B C))),E1,F1)],
                         E2,F2>)])])]
         <"(IMPR E0 (X) X),[λ<A,E1,F1> . F1¹(A)],E0,F0>),
        E2,F2>])
```

We can bind the context and arguments into this full significance:

```
= ([Δ("(EXPR E0 (X) (CAR X)))]                                          (S3-163)
   <"('(A B C)),E0,F0,
    [λ<S2,E2,F2> .
        <S2,
        ([Δ("(IMPR E0 (X) X))]
         <F0²("(QUOTE (A B C))),E0,F0,
          [λ<S2,E2,F2> .
               ([λ<S3,D3,E3,F3> . F3¹(D3)]
                <S2,
                 ([λ<A,E1,F1> . F1¹(A)]
                  <F0²("(QUOTE (A B C))),E0,F0>),
                 E2,F2>)]>),
        E2,F2>])
```

And perform the two CDRs off F_0 to pick up the arguments to QUOTE:

```
= ([Δ("(EXPR E0 (X) (CAR X)))]                                        (S3-164)
   <"('(A B C)),E0,F0,
    [λ<S2,E2,F2> .
        <S2,
         ([Δ("(IMPR E0 (X) X))]
          <"((A B C)),E0,F0,
           [λ<S2,E2,F2> .
               ([λ<S3,D3,E3,F3> . F3¹(D3)]
                <S2,
                 ([λ<A,E1,F1> . F1¹(A)]<"((A B C)),E0,F0>),
                 E2,F2>)]>),
         E2,F2>]>)
```

Next we need the internalised version of QUOTE from equation S3-142:

```
= ([Δ("(EXPR E0 (X) (CAR X)))]                                        (S3-165)
   <"('(A B C)),E0,F0,
    [λ<S2,E2,F2> .
        <S2,
         ([λS.λE.λF.λC . C(F¹(S),E,F)]
          <"((A B C)),E0,F0,
           [λ<S2,E2,F2> .
               ([λ<S3,D3,E3,F3> . F3¹(D3)]
                <S2,
                 ([λ<A,E1,F1> . F1¹(A)]<"((A B C)),E0,F0>),
                 E2,F2>)]>),
         E2,F2>]>)
```

Which we can then reduce:

```
= ([Δ("(EXPR E0 (X) (CAR X)))]                                        (S3-166)
   <"('(A B C)),E0,F0,
    [λ<S2,E2,F2> .
        <S2,
         ([λ<S2,E2,F2> .
              ([λ<S3,D3,E3,F3> . F3¹(D3)]
               <S2,([λ<A,E1,F1> . F1¹(A)]<"((A B C)),E0,F0>),E2,F2>)]
          <F0¹("((A B C))),E0,F0>),
         E2,F2>]>)
```

It is now straightforward to take the F_0 CAR: thus indicating that QUOTE returns its first argument. However, since we are aiming for the *designation* of (QUOTE (A B C)), this fact is ignored. What proves of interest is the designation of QUOTE, which is now applied to the original arguments:

```
= ([Δ("(EXPR E0 (X) (CAR X)))]                                        (S3-167)
   <"('(A B C)),E0,F0,
    [λ<S2,E2,F2> .
        <S2,
         ([λ<S3,D3,E3,F3> . F3¹(D3)]
          <"(A B C),
```

```
    ([λ<A,E1,F1> . F1¹(A)]<"((A B C)),E0,F0>),
    E0,F0>),
  E2,F2>]>)
```

We can now apply the designation of QUOTE as indicated. Note that S_3 and E_3, which have been brought along to establish the correct context, are at this point dropped; F_3 is used to do the CAR (in case an intervening RPLACA had actually modified the form under processing):

```
= ([Δ("(EXPR E0 (X) (CAR X)))]                                    (S3-158)
   <"('(A B C)),E0,F0,
    [λ<S2,E2,F2> .
        <S2,
         [F0¹([λ<A,E1,F1> . F1¹(A)]<"((A B C)),E0,F0>)],
         E2,F2>]>)
```

Another reduction:

```
= ([Δ("(EXPR E0 (X) (CAR X)))]                                    (S3-159)
   <"('(A B C)),E0,F0,
    [λ<S2,E2,F2> .
        <S2,[F0¹(F0¹("((A B C))))],E2,F2>]>)
```

Now we perform the inner of the two indicated CARs off F_0:

```
= ([Δ("(EXPR E0 (X) (CAR X)))]                                    (S3-160)
   <"('(A B C)),E0,F0,
    [λ<S2,E2,F2> .
        <S2,[F0¹("(A B C))],E2,F2>]>)
```

Thus we have shown that (QUOTE (A B C)) *designates* (A B C). The outer CAR (off F_0) is the explicit CAR from (CAR (QUOTE (A B C)); that we can do now:

```
= ([Δ("(EXPR E0 (X) (CAR X)))]                                    (S3-161)
   <"('(A B C)),E0,F0,
    [λ<S2,E2,F2> . <S2,"A,E2,F2>]>)
```

Finally, we have proved that, independent of what (CAR '(A B C)) *returns*, it *designates* the atom A (indicated in the meta-language by "A). We have not yet spelled out how the internaliser in this reformulation works, but its intent is clear, and the details can now be spelled out. This half of the derivation, however, will be quite brief, because some of the intermediate results are the same as ones we have already calculated. The internalisation of the primitive CAR closure we obtain from equation S3-138:

```
= ([λS.λE.λF.λC .                                                 (S3-162)
   [Σ(F¹(S),E,F,
      [λ<A,D1,E1,F1> . C([F1¹(A)],E1,F1)])]]
   <"('(A B C)),E0,F0,[λ<S2,E2,F2> . <S2,"A,E2,F2>]>)
```

Proceding with the reduction:

$$= [\Sigma(F_0^1("('(A\ B\ C))), E_0, F_0, \quad \text{(S3-163)}$$
$$[\lambda\langle A, D_1, E_1, F_1\rangle .$$
$$([\lambda\langle S_2, E_2, F_2\rangle . \langle S_2, "A, E_2, F_2\rangle] \langle [F_1^1(A)], E_1, F_1\rangle)])]$$

And:

$$= [\Sigma("(QUOTE\ (A\ B\ C))), E_0, F_0, \quad \text{(S3-164)}$$
$$[\lambda\langle A, D_1, E_1, F_1\rangle .$$
$$([\lambda\langle S_2, E_2, F_2\rangle . \langle S_2, "A, E_2, F_2\rangle] \langle [F_1^1(A)], E_1, F_1\rangle)])]$$

But of course we have already gone through the determination of the full significance of (QUOTE (A B C)) in S3-151 through S3-159, above. Admittedly we have a different continuation this time, but the computation is the same. Thus we can step immediately to the result:

$$= ([\lambda\langle A, D_1, E_1, F_1\rangle . \quad \text{(S3-165)}$$
$$([\lambda\langle S_2, E_2, F_2\rangle . \langle S_2, "A, E_2, F_2\rangle] \langle [F_1^1(A)], E_1, F_1\rangle)]$$
$$\langle "(A\ B\ C), $$
$$([\lambda\langle A, E_1, F_1\rangle . F_1^1(A)]\langle "((A\ B\ C)), E_0, F_0\rangle),$$
$$E_0, F_0\rangle)$$

This time when we substitute we ignore the designation, and concentrate on what was returned:

$$= ([\lambda\langle S_2, E_2, F_2\rangle . \langle S_2, "A, E_2, F_2\rangle] \langle [F_0^1("(A\ B\ C))], E_0, F_0\rangle) \quad \text{(S3-166)}$$

There is one final CAR to be performed in F_0:

$$= ([\lambda\langle S_2, E_2, F_2\rangle . \langle S_2, "A, E_2, F_2\rangle] \langle "A, E_0, F_0\rangle) \quad \text{(S3-167)}$$

And finally we can apply the final continuation:

$$= \langle "A, "A, E_0, F_0\rangle \quad \text{(S3-168)}$$

We are done. We have proved that the expression (CAR '(A B C)) both designates and returns the atom A, without side effects. As expected.

We will not trouble with more examples; all that remains to reconstruct our previous machinery in this new formulation is to define new versions of EXT and INT, and show how they would be used. In particular, we noted that Σ of the primitive addition procedure was as follows:

$$\Sigma(E_0 F_0("+")) = \quad \text{(S3-169)}$$
$$\lambda E.\lambda F.\lambda C .$$
$$[C("(EXPR\ \underline{E0}\ (B\ C)\ (+\ B\ C)),$$
$$[\lambda\langle A, E_1, F_1\rangle .$$

```
                Σ(F₁¹(A),E₁,F₁,
                   [λ<A₂,D₂,E₂,F₂> .
                      Σ(F₂¹(F₁²(A)),E₂,F₂,
                         [λ<A₃,D₃,E₃,F₃> . +(D₂,D₃)])])]
E, F)
```

It is the second argument to the continuation that is in question: it would be easier if we could have said:

```
Σ(E₀F₀("+)) = λE.λF.λC .                                        (S3-170)
                  [C("(EXPR E0 (B C) (+ B C)),
                     EXT(+)
                     E, F)]
```

These considerations suggest the following definition of EXT. It differs from the previous one in just the way we would expect: rather than simply referring to the designation of the arguments in the context of use of the whole, it iteratively steps through the full significance of each argument, so as to deal effectively with side effects, but in the end applies the original function to the set of designations returned.

```
EXT ≡ λG.[λ<A,E,F> .                                            (S3-171)
            Σ(F¹(A),E,F,
               [λ<B₁,D₁,E₁,F₁>.
                  Σ(F₁¹(F²(A)),E₁,F₁,
                     [λ<B₂,D₂,E₂,F₂>.
                        Σ(F₂¹(F₁²(F²(A))),E₂,F₂,
                           ...
                              [Σ(F_{k-1}¹(F_{k-2}²(...(F₁²(F²(A)))...)),E₂,F₂,
                                 [λ<Bₖ,Dₖ,Eₖ,Fₖ>. G(D₁,D₂, ... ,Dₖ)])]...)])])]
```

Note the use of different fields in each of the CDRs, as each argument is extracted, reflecting the fact that the processor steps down the argument list, in such a way that side-effects to that list before (i.e., closer to the head than) the current argument position do not affect the processor's access to subsequent arguments.

The corresponding definition of INT is far simpler, of course, because the arguments are not processed:

```
INT ≡ λG.[λ<A,E,F> . G<F¹(A),F¹(F²(A)),...,F¹(F²(...(F²(A))...))>]      (S3-172)
```

Thus for example we have the following full significance of QUOTE:

```
Σ(E₀F₀("QUOTE)) = λE.λF.λC .                                    (S3-173)
                     [C("(IMPR E0 (X) X), INT(λX.X), E, F)]
```

It would be convenient if we could similarly define an EXPR and IMPR as meta-theoretic functions that would generate the internalisations automatically. In 2-LISP we will be able to do this, because we will have a notion of normal-form available in which to cast the answer. For the present, however, lacking such apparatus, we would have to define the internalisations individually.

3.e.v. The Evaluation Theorem

In spite of the extent of the explorations of section 3.e.ii, we are far from done. Nonetheless, we have spent as much time on semantic characterisation as we can afford, given our long-range goal of reflection (and we have probably asked as much patience of the reader as can reasonably be expected). It should be clear, however, that we have, in outline at least, accomplished our main task: we have provided a mechanism whereby the full significance of the primitives can be defined, and we have shown how the significance of composite structures derives from the significance of the parts. We have indicated as well how both declarative and procedural import are carried by this full significance, in a partially related, but not identifiable, fashion. Sufficient distinction between them remains so that we can examine, for any given expression, the relationship between its result and its referent.

If we were to proceed in this fashion, setting out the primitive significance (and internalisation) of all the primitive procedures (we have already done this for the structure types), we would of course see that EVAL — the projection of LISP's Σ onto its first coordinate — in some cases dereferences its argument, and in some cases does not. It is natural to ask, when one first encounters this fact, whether there is "method to EVAL's madness": whether there is any lurking fact that determines when EVAL dereferences and when it does not. The answer — obvious given our long exposition, but not when one first considers the situation — is a clear "yes": evaluation in LISP is de-referencing *just in case the referent is in the structural field*: LISP's evaluator *dereferences if it can*, and *simplifies* otherwise. This is the observation we have called the evaluation theorem, which by rights we now should prove.

If we were to set out on that project, we would adopt the following strategy. First, we would define as ***standard*** any 1-LISP procedure with the following property: all

applications in terms of it satisfy the theorem. The intent is carried by the following informal characterisation:

```
STANDARD(S) ≡ for [S1 = Γ"(S ... )7]                                (S3-174)
                  if [ΦEF(S1) ∈ S]
                     then [ΨEF(S1) = ΦEF(S1)]
                     else [ΦEF(ΨEF(S1)) = ΦEF(S1)]
```

This can more properly be put as:

```
STANDARD : [ S → {Truth, Falsity} ]                                 (S3-175)
  ≡ λS ∈ S .
      [∀P ∈ PAIRS, E ∈ ENVS, F ∈ FIELDS
         [[F¹(P) = S] ⊃ [if [ΦEF(P) ∈ S]
                            then [ΨEF(P) = ΦEF(P)]
                            else [ΦEF(ΨEF(P)) = ΦEF(P)]]]]
```

For example, any procedure that *designated* a function whose range was entirely within the structural field, and whose computational significance was such that any application in terms of it would return its referent, would be called standard. In addition, any procedure that designated a function whose range was *outside* the structural field, whose significance was correspondingly such that any application in terms of it would return a designator of its referent, would also be standard. CAR and QUOTE, for example, are (bound in the initial environment to) standard procedures for the first reason; + is similarly standard, for the second reason. However a procedure can be standard *even if its designated range cannot be classified as either in or outside of S.* The conditional IF, for example, returns the result of one of its second or third arguments, depending on the first: `(IF T 1 'A)` designates the number one and returns a co-designative numeral; `(IF F 1 'A)` designates the atom A and returns that atom, for example. Thus the range of IF includes all of *D*, not just *S* or its complement. Nonetheless, IF is standard in the sense just defined.

The proof would proceed first by showing that the atomic structure types obey the evaluation theorem — this we have essentially shown already. Then we would show that all the primitive procedures are standard. This is not quite as simple as it might seem; it is immediate that CAR, for example, returns an object within the field, and designates an object within the field, but it is not so immediately clear that it will always be the *same* object. We would have, for example, to examine the primitively provided internalisation of CAR, since that is implicated in determining the local procedural consequence of CAR applications. In other words we would have to show the compatibility of the following two

equations defining CAR:

$$\Sigma(E_0F_0(\text{"}CAR)) = \lambda E.\lambda F.\lambda C\ .\ [C(\text{"}(EXPR\ \underline{E0}\ (X)\ (CAR\ X)),\ [\lambda\langle A,E_1,F_1\rangle\ .\ \Sigma(F_1^1(A),E_1,F_1,[\lambda\langle S_2,D_2,E_2,F_2\rangle\ .\ F_2^1(D_2)]])]\ E,\ F) \qquad \text{(S3-176)}$$

and:

$$\Delta E_0F_0(\text{"}CAR) \equiv \lambda S.\lambda E.\lambda F.\lambda C\ .\ [\Sigma(F^1(S),E,F,\ [\lambda\langle A,D_1,E_1,F_1\rangle\ .\ C([F_1^1(A)],E_1,F_1)]] \qquad \text{(S3-177)}$$

It is not immediate that these imply that $\Psi EF(\text{"}(CAR\ \ldots\ )) = \Phi EF(\text{"}(CAR\ \ldots\ ))$. One strategy that might be of help would be to define a function STANDARD-PRIMITIVE that would simply *assert* the above two characterisations, given two inputs: it could then be used to *define* the primitive import of various of the provided procedures.

Though involved, this could presumably be done. Once the primitives had been proved standard, we would then demonstrate (using recursion induction) that all compositions and all abstractions definable in terms of the primitives were also standard (by looking at the full significance of arbitrary closures, and showing that procedural import, designation, and internalisation all worked in step so as to preserve "evaluation" properties). It would then be immediate that the dialect as a whole satisfied the theorem, because a proof that pairs satisfied it would follow directly from the fact that the term in procedure position must be standard.

We will, however, not do this here; we leave it — to employ the standard dodge — as an excercise for the reader. Note, however, that the approach is not specific to this particular theorem; the same technique could be used to show that 2-LISP satisfies the normalisation theorem — i.e., that designation is always preserved — given a different notion of what counts as being *standard.* It is with this strategy in mind that 2-LISP will be presented in the next chapter.

3.f. Towards a Rationalised Design

What then have we learned from this analysis of 1-LISP? First, we have seen that the lack of category alignment between the typology of the structural field and the categories of semantic consequence is bothersome. Furthermore, the inelegance that resulted did not simply make programs more complex: it mandated certain uses of meta-structural machinery that were not strictly necessary (that is why a matter of aesthetics is of such concern to us). This can readily be repaired in a new design. We have suggested as well that the de-referencing behaviour implied in the received notion of evaluation is problematic, that a revised dialect should be based on a computable function Ψ that is designation-preserving, and that that declarative interpretation function should be defined without recourse, explicit or implicit, to the mechanism used to compute Ψ. From this suggestion emerged the suggestion that Ψ take expressions into normal form.

There are two questions that still deserve attention, before moving to the design of 2-LISP. The first has more to do with the relationship between evaluation and reference. It might seem, by this point in the analysis, that our stand against evaluation would long since have been taken. We seem to have shown, as summarised in the theorem bearing its name, that evaluation conflates issues of expression simplification and term de-referencing. Indeed, this is our position, but we have not yet *defended* it: all that we have demonstrated is that there is a particular *correlation* between evaluation and reference (providing you accept our account of the referential import of LISP expressions). But so far the analysis has been symmetric: one could equally well conclude, if we did not examine the issue further, that *simplification* and *de-referencing* were two somewhat related, and rather partial notions, each ineffectually covering a piece of the far more natural concept of evaluation.

The notion of evaluation is by all accounts a natural notion of computer science; the concepts of simplification and reference we have used here are borrowed without apology from logic, mathematics, and philosophy. Another way to make the point of the last paragraph, therefore, is to say that we have shown only how the theoretical categories of two disciplines relate, in a particular instance. We have not, in other words, provided sufficient ammunition to enable us to *choose* one as better. In section 3.f.i, therefore, we will focus on the concept of evaluation in its own right, in an attempt to lay to rest any

lingering feelings that we are treating it unfairly. It is one goal of this dissertation to convince the reader that the very concept should be struck from our theoretical vocabulary; we will therefore make plain our final position.

Secondly, we have not yet said very much about what the concept of normal form should come to, except to remark that the definition of this notion used in the λ-calculus — based, essentially, on the concept of not being further β-reducible — fails to meet our standard of category identifiability. We mentioned at the outset that there were various properties that should be associated with any normal-form designator — that of being context independent, side-effect free, and stable — but we have said nothing about how a computable notion of normal form is to be defined. It will thus be appropriate to examine this notion further in section 3.f.ii, since once we have an appropriate definition in hand we will be sufficiently equipped to set out on the design of 2-LISP.

In chapter 2 we embedded a simple theory of LISP within LISP, by constructing a meta-circular processor. As became much clearer in the more detailed analysis of the present chapter, that theory was of the procedural import of LISP s-expressions. In the current chapter we constructed another theory of LISP, this time encoded in a λ-calculus meta-language, of both the declarative and procedural import of the elements of the structural field. In chapter 5 we will erect our third meta-theoretic characterisation of LISP, this time within the code of the 3-LISP reflective processor. That third characterisation will in some ways be like the 1-LISP meta-circular processor, and in some ways like the meta-linguistic accounts presented here in chapter 3. In section 3.f.iii we will review a variety of the features of our meta-theoretic account that, although they did not merit mention while we were in the midst of describing 1-LISP, will turn out to be important when we take up the reflective goals.

Finally, the chapter will end in section 3.f.iv with a short discussion — included by way of a footnote — on data abstraction. It will have occurred to the reader that our considerable emphasis on the declarative import of atomic and composite structures would seem to fly in the face of the received wisdom that one should define data structures behaviourally, without regard to the structures in terms of which they are implemented. Indeed, the tension between our declarative stance and the behavioural (instrumental) cast of the procedural tradition is strong, and deserves at least some comment. Although we

will argue that the two positions are not fundamentally opposed, the apparent conflict between them will have to be explicitly defused.

3.f.i. Evaluation Considered Harmful

The evaluation theorem simply states a formal relationship: it does not, and cannot, itself bear normative weight. The critique of evaluation requires further argument. In particular, we will reason as follows: if we had an independently definable notion of evaluation — a pre-theoretic or lay intuition that this formal concept was intended to capture, or a concept playing such a cornerstone role in some theoretical structure that its utility could not lightly be challenged — then we might be able to argue from first principles for what the value of any given expression should be. Subsequently, if a formal mechanism were proposed that was claimed to effect an evaluation mechanism, then we could perhaps prove that this mechanism indeed embodied the independently formulable notion of evaluation.

The problem, however, is that we have no such independent notion of evaluation. At least we have no *formal* notion: to the extent that there seem to be pieces of a natural concept, they are not formal notions, and therefore evaluation cannot be something that any formal processor can itself effect. The structure of the argument should be clear. *First* is the recognition that computation is based foundationally on semantical attribution. *Second* is the claim that, because of this, it is important to establish that attribution independently of the procedural treatment of formal structures. *Third* is an aesthetic claim that, once this attribution is set forth, the procedural treatment should emerge as semantically coherent, in terms of the prior account. Given this structure, we challenge evaluation in a double manner. We are *not* claiming that it is incoherent as a procedural regime — in fact it is self-evidently tractable. LISP, after all, has survived unchallenged for two decades; in addition, we expended considerable effort in the previous sections to characterise it precisely. Rather, the claim is that *if evaluation is taken as a procedural regime*, it fails to cohere with the prior attribution of significance. Alternatively, *if it is claimed to be an independent notion*, then the received understanding of it fails to be evaluation. We will look at these two options in turn.

The basic problem — common both to evaluation and to *application*, which we will subject to the same scrutiny — is one of distinguishing use from mention: in employing these terms, do we refer to abstract mathematical entities, or to structures that signify those entities? For example, do we want to say that we apply the *mathematical addition function* to two *numbers*, or do we want to say that we apply an *expression* that designates that function to two *expressions* that designate numbers? Both ways of speaking are coherent, but we cannot use the same term to refer to such crucially distinct circumstances.

Historically, there seem to be three standard uses of the terms "value" and "evaluation", stemming from mathematical and logical traditions. One has to do with functions, and is involved with the use of the term *application*: a paradigmatic use of the term "value" is with regards to a function: the *value* of a function *applied* to an argument is the element of the range of the function at that argument position. Thus the *value* of the addition function, applied to the numbers 2 and 3, is the number 5. Similarly, the value of the square-root function applied to the number 169 is the number 13. The usage is a little strange, since it is not quite clear whether it is the *function* that has an argument-relative value, or whether there is an abstract application consisting of a function and arguments, that possesses the value. From an informal standpoint, however, such terminology is clear enough, and we will continue to use the term — with caution — in such a circumstance.

A second, only partly related use of the term "value", and one that engenders far more confusion, has to do with *variables*. If any particular variety of object has an unchallengeable claim to having a *value*, it would seem to be a variable. Thus we may ask what `(+ X Y)` is, if the *value of* `X` *is three*, and if the *value of* `Y` *is four*. Finally, a third notion of value, perhaps an extension of the foregoing usage, has to do not with particular variables but with whole expressions. In mathematics, for example, it seems uncontroversial to evaluate an expression, like `X + (Y * Z/3)`. This expression, if `X` is `2`, `Y` is `10`, and `X` is `15`, would be said to have a value of `62`. Similarly in first order logic, a sentence like `∃X [MORTAL(X) ∧ SAD(X)]` might be said in a particular world to have a *value*. In fact the very use of the term "truth-value" betrays this assumption.

In both of these last two cases it is clear that the *value* of the variable or expression is the *referent* or *designation*: the *value*, in other words, refers to what the term *denotes*. In the mathematical examples, the value of the variable `X` was assumed to be the real

(Platonic, whatever) number two, not the *numeral* 2. This is incontestable: if the value were the numeral, it would make sense, on being asked what the value of X + Y is when X is 2 and Y is 3, to reply that the answer depends on whether one is using Arabic or Roman numerals. This is crazy: the value is 5 independent of symbology precisely *because* the value is the *number*, not a sign designating that number. Similarly, if we said that the open sentence [MORTAL(X) ∧ SAD(X)] was satisfied by Socrates, then the value of the existential variable is the philosopher himself, not a designator or name.

This referential or designational sense of "value" is reflected in the use of the phrase "valuation functions" for what we are calling interpretation functions: the main semantical functions that map signs onto significants. The same referential sense is reflected in Quine's dictum that "*to be is to be the value of a bound variable*"[15] (a maxim that accords well with our definition of an object as the referent of a noun phrase). In sum, to say that Y is the value of X is to imply not only that X is a sign but that it is a *term*, and that Y is *the object designated by that term.*

This conclusion immediately raises trouble about the proper use of the term "evaluation" in a computational context. It seems established that evaluation must be a process defined over signs, but if evaluation is a *function* it would seem that it should return the *value* of its argument, implying that *evaluation must dereference its argument.* This can be put more strongly: to evaluate *is to* dereference, on the standard reading. It is of course possible that the value of a sign may itself be a sign (since signs can be part of a semantical domain), but it nonetheless follows that no expression in a formal system can properly be said to be evaluated that designates an abstract entity such as a number or function, or an external object like a person or table.

No computer, in other words, can evaluate the expression "(+ 2 3)".

Some readers may object to this claim. A possibly reply to it that might be offered to counter our objections — one we might expect to hear in computational circles — is that there is no problem having a computer evaluate "(+ 2 3)" if we take numerals to be self-referential in just the sense that we saw numerals to be "self-evaluating" in 1-LISP. Certainly this is mathematically tractable: no special problems are raised in the mathematical model theory by having certain objects be their own referents, as any number of semantical accounts have shown. The problem is much simpler: as we have said before,

to say that numerals refer to themselves is *false*. As we have held all the while, we are not simply attempting to erect *some* mathematically coherent structure: we are attempting to make sense *of our attribution of reference to formal symbols*. There is simply no possible doubt that every computationalist takes the numerals `1`, `2`, `3`, and so forth to designate numbers — it is almost impossible to overcome the natural tendency to do so. This is made self-evident by the fact that the operations we define over them we call addition, multiplication, etc. Anyone who attempts to hold the view that numerals are their own referents must suffer the embarassment of admitting that the expression `(+ 2 3)` has nothing to do with addition, since addition is defined over numbers, not over numerals. Such a person would then have to claim that he or she is using the word "addition", as well as "reference", in other than their normal sense. Such a person, in other words, is forced gradually to sever any connection between what we *claim* the machine is doing and how we *understand* what the machine was doing. The only possible result of such an approach is the kind of confusion we are trying to rectify: a dissonance between our natural understanding of computation and our formal analysis of computational systems. In sum, there is simply no tenable retreat to be found in this direction.

One would have in addition to reject all the claims of the standard denotational semantics accounts saying that LISP procedures naturally designate functions, since the notions of value and evaluation *in the meta-languages employed in those semantical endeavours* are the notions we have just endorsed — the extensional, referential ones — not the computational ones.

Nor is there solace to be found in a position that says that computers can access actual Platonic numbers (whatever they might be) as well as numerals, since that violates the fundamental notion of computation.

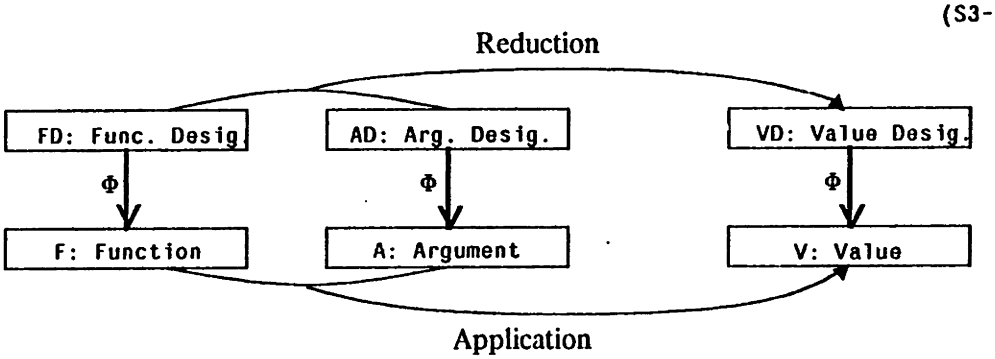
In passing, note that nothing is being said about whether *people* can evaluate an expression such as `(+ 2 3)`: it is by no means clear what it is to say of a person that he or she evaluates an expression, since it is not clear whether to do this is to compute a function in any sense, or whether there is any salient notion of *output* or *result* of such a process. Fortunately, such questions do not need to be answered in this context. What we are left with, however, is the conclusion that we must be more careful in describing what `1-LISP` does. We will not have to change its behaviour: what is under attack, it should be clear, is

our *account* of that behaviour. We are not saying, in other words, that there is any problem in typing "(+ 2 3)" to the 1-LISP interpreter and having it type "5" in response: rather, we are saying that in doing that 1-LISP cannot be said to be returning the *value* of the input expression.

The situation regarding function application is only slightly more complex than evaluation, because it relates three, rather than two, objects of some variety. The trouble arises over whether application is a function defined over abstract functions, or over expressions. Informally, the idea is that a function is applied to its arguments to determine a value: from this *application* would seem to be a relationship between functions, arguments, and values. This is the usage we just agreed to maintain regarding values, so it is natural to use application in the same way. The consequence, however, is that application is not what APPLY in 1-LISP does, since that function is defined over expressions, not over abstract functions.

The situation will be made clear by considering the following diagram:

(S3-178)



In the terms of the figure, we are asking whether one applies F to A to yield V, or whether one applies FD to AD to yield VD. Consider an expression such as "(PLUS 3 4)". The question we need to resolve is whether the *expression* "PLUS" is applied to the *expression* "(3 4)", yielding the *expression* "7", or whether the *addition function* is applied to the *two numbers* 3 and 4, yielding the *number* 7.

Both concepts, of course, are coherent: we have agreed to use application for the latter, implying that we need a term for the former — a term to take the place of the APPLY of 1-LISP. What we will strictly want to avoid, however, in an effort to maintain at least a modicum of clarity on this subject, is using terms that *cross semantical levels*, such as

among F, AD, and VD. It should be clear that in 1-LISP the function APPLY designates a relationship among FD, AD, and V; in SCHEME, the same name is used to designate a relationship among F, AD, and V.

In the remainder of this dissertation we will adopt the following definitions. ***Application*** will be taken to be a three-place relationship among an abstract function, an argument (or arguments), and what is called the value of that function at that argument. For example, we will say that the *addition function* applied to the *numbers 2 and 3* will yield the *number 5.* By "application", in other words, we refer to the relationship in the lower part of S3-178, among F, A, and V.

The other relationship — among FD, AD, and VD, we will call ***reduction***, in part because of its relationship to the β-reduction of Church, and also because the term connotes a relationship among expressions or linguistic objects, rather than between arbitrary objects in the world (even its use in philosophy of science as between one theory and another is compatible in spirit). Of the two "reductions" in the λ-calculus, it is β-reduction that actually "reduces" the complexity of a lambda term; α-reduction is not particularly a "reduction" in any natural sense of that term. It is not always the case that reduction in the LISPs we will examine reduce complexity, because names are looked up, which can increase the apparent complexity. If, however, one takes the complexity of an expression to include the complexity of the bindings of all the variables occuring within it, reduction in this new sense is in point of fact reductive of complexity in the general case.

Thus we will say that the function designator "+" and the argument designator "(2 3)" reduce in 1-LISP to the numeral "5", although it should be straight away admitted that we have so far not uniquely defined reduction, since we have said nothing about what expression a composite expression should reduce to, except that it should designate the value of the function at that argument position. In other words, by the characterisation just given + and (2 3) might reduce to (+ 2 3), since the latter term designates, tautologically, the value of the addition function applied to the numbers two and three. However we will of course use the term "reduction" to relate a function designator, an argument designator, and a *normal-form* designator of the value of that function applied to those arguments.

The reduction function, which will in 2- and 3-LISP be designated in the initial environment by the term REDUCE, will play a considerable role in our considerations of

reflection. We will not, however, make use of functions called APPLY or EVALUATE, for the simple reason that, on the readings we have just given to those terms, they are entirely unnecessary. In particular, any expression of the form (this is 2-LISP syntax):

```
(APPLY F A)                                                    (S3-179)
```

is entirely equivalent to a simple use of the terms, in the following type of expression:

```
(F . A)                                                        (S3-180)
```

Thus we could define APPLY in 2-LISP as follows (for lists of formal parameters, 2-LISP brackets are like 1-LISP parentheses):

```
(DEFINE APPLY                                                  (S3-181)
   (LAMBDA EXPR [FUN ARGS] (FUN . ARGS)))
```

It would follow, in addition, since application is a higher-order function, that any number of applications of APPLY would all be empty. The following, in particular, would all be declaratively and procedurally equivalent (here the brackets should be treated as enumerators -- thus 2-LISP's (PROC A [B C]) is syntactically not unlike 1-LISP's (PROC A (LIST B C))):

```
(FUN . ARGS)                                                   (S3-182)
(APPLY FUN ARGS)
(APPLY APPLY [FUN ARGS])
(APPLY APPLY [APPLY [FUN ARGS]])
...
(APPLY APPLY [APPLY [APPLY [ ... [APPLY [FUN ARGS]]...]]])
```

REDUCE, on the other hand, since it is a meta-structural function, is neither trivial to define, nor recursively empty. In particular, whereas

```
(+ 2 3)                                                        (S3-183)
```

would simplify to the numeral 5, the expression

```
(REDUCE '+ '[2 3])                                             (S3-184)
```

would simplify to the numeral *designator* '5. Similarly, a double application of reduction, of the following sort:

```
(REDUCE 'REDUCE '['REDUCE '[2 3]])                             (S3-185)
```

would simplify to the double designator ''5. Furthermore, such meta-structural designation is necessary in order to avoid semantical type errors: the following expression would

generate an error:

```
(REDUCE '+ [2 3])                                    (S3-186)
```

since reduction is defined over expressions, and the second argument in this case designates an abstract sequence of numbers.

All of these issues will become clearer once 2-LISP is introduced.

3.f.ii. Normal Form Designators

Our aesthetic mandate required that the *category* of a normal-form designator depend solely on $\Phi(S)$ — this was what we called the semantical type theorem. We noted at the outset that, if functions are to be first class objects in the semantical domain, we cannot hope to achieve a notion of *canonical* normal form, since that would require that we be able to inter-convert all expressions designating the same mathematical function, which is evidently non-computable. Hence we must expect that, whatever notion of normal form we adopt, functions on the standard interpretation will possibly have multiple normal form designators. There is no *harm* in this — it does not weaken the LISP that results in any way — it is merely worth admitting straight away.

Our approach to defining a generally adequate notion of normal form will be to look at the various kinds of element in our domain D. For the number and truth-values the obvious normal form designators are the numerals and the two boolean constants. These are canonical, and are what Ψ for 1-LISP took designators into, so they are highly recommended. For s-expressions we also have an obvious suggestion: their quoted form. There is a minor problem here, regarding uniqueness: in 1-LISP, as we noted earlier, a given s-expression can have distinct quotations:

$$\forall S_1 \in \mathcal{S},\ S_2 \in \mathcal{S}\ [[S_1 = S_2] \supset [\ulcorner \text{(QUOTE } \underline{S_1}\text{)} \urcorner = \ulcorner \text{(QUOTE } \underline{S_2}\text{)} \urcorner]] \qquad \text{(S3-187)}$$

However we have a solution to this already mandated by the category alignment aesthetic. The requirement that the *structural* types correspond to *semantic* types in as clear a fashion as possible has been satisfied for the *numbers* and *truth-values*, providing we make the two boolean constants distinct from regular atoms. Thus if we make s-expression designators a unique structural type — we will call them *handles* — we can simply establish by *fiat* that all handles that designate the same s-expression are themselves the same handle. Thus two problems are solved with one solution.

There are two other segments of the semantical domain to be treated: the user's world of objects and relationships, and the set of continuous functions. About the first — to be designated only by base-level structures and not (tautologically) by any terms in *programs* — we have little to say here, except what is laid out in the following section on data abstraction. The other semantical entities are the functions.

Before considering them, note that all suggestions for normal-form designators made so far, in terms of the Ψ of the preceding sections, satisfy the three constraints we mentioned earlier regarding such normal forms: they are *environment-independent*, *side-effect free*, and *stable*. Formally, this can be stated as follows, if we define the notion of ***normal-form*** in terms of these three properties:

$$\texttt{CONTEXT-IND} \equiv \lambda S\,[\,\forall E_1,\ E_2 \in \textit{ENVS},\ F_1,\ F_2 \in \textit{FIELDS} \quad [[\,\Phi E_1F_1(S) = \Phi E_2F_2(S)\,] \wedge [\Psi E_1F_1(S) = \Psi E_2F_2(S)\,]]] \tag{S3-188}$$

$$\texttt{SE-FREE} \equiv \lambda S\,[\,\forall F \in \textit{FIELDS},\ C \in \textit{CONTS},\ E \in \textit{ENVS} \quad [\,\Sigma(S,F,E,C) = C(\Psi EF(S),\Phi EF(S),E,F)\,]] \tag{S3-189}$$

$$\texttt{STABLE} \equiv \lambda S\,[\,\forall F \in \textit{FIELDS},\ E \in \textit{ENVS}\,[\,S = \Psi EF(S)\,]] \tag{S3-190}$$

$$\texttt{NORMAL-FORM} \equiv \lambda S\,[\,\texttt{CONTEXT-IND}(S) \wedge \texttt{SE-FREE}(S) \wedge \texttt{STABLE}(S)\,] \tag{S3-191}$$

The result we have already is this:

$$\forall S \in \textit{S}\,[[\,S \in \textit{NUMERALS} \cup \textit{BOOLEANS} \cup \textit{HANDLES}\,] \supset \texttt{NORMAL-FORM}(S)\,] \tag{S3-192}$$

What will remain, of course, is to prove the normalisation theorem in the general case:

$$\forall S \in \textit{S},\ E \in \textit{ENVS},\ F \in \textit{FIELDS} \quad [[\,\Phi EF(S) = \Phi EF(\Psi EF(S))\,] \wedge [\,\texttt{NORMAL-FORM}(S)\,]] \tag{S3-193}$$

As mentioned earlier, the compositional aspects of such a proof can be handled by straightforward techniques; the present question is how we deal with functions.

It is at this point where the notion of a closure suggests itself as the reasonable candidate for a normal-form function designator. We commented in chapter 2 that it was unclear whether a closure was an s-expression or not: it was clearly a finite object — not, in other words, a real *function* in the sense we are using that term — and, since it embeds its defining environment with in it, it is an environment-independent object (if applied in any environment it will yield the same function). We commented that one reason it may not be considered to be a valid *expression* is that it doesn't have any obvious *value*, but of course

this is a criticism that has by this point evaporated.

1-LISP closures, as we remarked in chapter 2, are distinguished combinations having special atoms — EXPR and IMPR — in "function position". We can thus posit the following: a combination formed of the atom EXPR or IMPR followed by normal form designators of an environment, a variable list, and a "body" expression will be defined to be normal-form designators of functions. Thus for example:

```
∀f ∈ ENVS, F ∈ FIELDS                                          (S3-194)
    [[[ ΨEF("(LAMBDA EXPR (X) (+ X 1))) ]
        = [ Γ"(EXPR '(X) ENC(E) '(+ X 1))⌝ ]] ∧
     [ NORMAL-FORM(ENC(E)) ]]
```

Two questions are raised by this example: what it is to be a normal-form designator of an environment (and whether environments will constitute an addition to the semantical domain), and what functions the terms EXPR and IMPR designate, since we are using them in the function position of a procedure application.

Regarding the first question, we have treated environments as functions in our mathematics: there is no immediate reason to do so differently *within* the calculus. From this stand it follows that the normal-form environment designator is recursively defined by the above equation. However there is something odd about this: since all closures contain environment designators within them, it would seem that environment designators would be circularly defined. It is also true, however, that environments do not need access to an enclosing environment designator; thus we could for example posit that environment designators contain themselves as their own enclosing environment (i.e., the second element of an environment closure would be the closure itself).

In fact, however, we will adopt a different strategy, in part to make it easier for environment designators to be *modified* in a reflective dialect. Note that the 2-LISP enumerating structure (called a rail) designates a sequence; it follows that a rail of two-element rails will designate an *ordered* set of two-element tuples. Functions, on the other hand are *unordered* sets of two-element tuples. The referents of this particular kind of rail, in other words, and of closures, are very close in structure. Since procedure application is defined as *extensional in first position* — that of the function designator — we are committed to allowing the function to be designated by any type of syntactic expression; thus if braces were legal 2-LISP notation used to notate a new data type designating sets —

a data type called, say, a ***sack*** — they too could be used in function position in a procedure application. In other words, suppose that the expression {1 2 T} designated the three-element set consisting of two numbers and a truth value. Then equation S3-135 would seem to require that

```
({[2 20] [4 50]} (+ 1 3))                                    (S3-195)
```

designate fifty, and return the numeral 50. This is mandated because, although sacks and closures are *intensionally* distinct, *extensionally* they are equivalent. We would have a problem in our meta-theory, to deal with this case, because the internaliser Δ is currently defined only over closures, not over sacks (assuming we posit sacks as *normal-form* designators of sets). On the other hand, we would have other problems as well: sacks and closures would come into conflict as normal-form designators for sets of a certain structure, thus violating the semantical type theorem.

We do not have to solve this problem, because we do not have such a structural type, but it does lead to the following suggestion. We can *extend* the definition of the behaviour of sequences of a certain structure so that they can be applied, as if they were functions, with the additional constraint that if more than one tuple has the argument as its first element, then the tuple closer to the front of the sequence is used to determine the value of the application. Sequences, then, can be applied directly to obtain a kind of "association-list" behaviour that is often primitively provided in LISPs. For example, the expression

```
([[2 20] [(* 2 2) 50] [4 'HELLO]] (+ 1 3))                   (S3-196)
```

would designate fifty, and return 50.

Given such a protocol, we can define an ***environment*** to be an *ordered* sequence of tuples of atoms and bindings. Our meta-theoretic characterisation still holds, since these can still be applied, and since rails are the normal-form designator of sequences, we are given an answer as to what form a normal-form designator of an environment will take. In the reflective code we can use such applications as (ENV VAR) to designate the binding of the atom designated by VAR in the environment designated by ENV. In addition, since environments will not be *normally* designated with closures, the circularity problem just adduced does not arise. Finally, rails are eminently suitable ground for side-effects, facilitating the requirements of reflection. Thus a variety of concerns can be dispensed with

rather easily.

The closure, then, of, for example, the designator (LAMBDA EXPR (X) (+ X 1)), would have the following form:

```
(EXPR [[VAR1 BINDING1] [VAR2 BINDING2]... [VARk BINDINGk]]          (S3-197)
      '[X]
      '(+ X 1))
```

There is however one final problem with this solution. This closure is a pair; by standard assumption it must designate the value of the function designated by the CAR applied to the arguments designed by the CDR. The *arguments* are all normal form designators, but the *procedure* is named by the atom EXPR; we need to ask what function it designates, and how its name can be normal-form.

As to what function it designates, that is straightforward: it can signify an *extensional* procedure (even though LAMBDA is intensional) that takes environments, sequences of variables, and a body expression, onto the function designated. *EXPR*, in other words, *designates the TRANS* function of section 3.d. I.e., we have (in the initial context E_0 and F_0):

$$\Phi E_0 F_0(\text{"}EXPR) = EXT(TRANS) \qquad \text{(S3-198)}$$

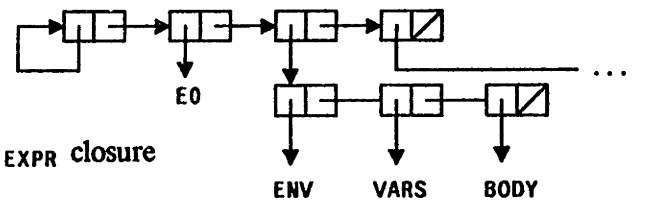
The question about EXPR's name appearing in the procedure position of closures is, however, trickier. Closures must be stable; on the other hand we must support such code as:

```
(LET [[EXPR +]] (EXPR 2 3))                                          (S3-199)
```

This clearly must designate five and return the numeral 5. Thus we simply cannot have EXPR appear as an atom in function position. No atoms are in normal form; thus no atom can appear in the procedure position of a closure. Hence we must reject S3-197.

The obvious suggestion is that the *normal-form* of the procedure signified by EXPR should appear as the CAR of each extensional closure. This would seem to be a circular requirement, since the question re-appears in asking what appears in the CAR position of the EXPR closure. However it is not a circular *account*, which would be vicious, but rather a requirement for a *circular structure*, which has a simple answer: the EXPR closure should be its own CAR. Thus the structure of the EXPR closure, though it is not lexically printable, is notated as follows (in fact this is 1-LISP graphical notation for 1-LISP structure; we will eventually adopt a 2-LISP analogue of this structure):

(S3-200)



Since EXPR is primitive, this raises no particular problems.

This much of a discussion should indicate that a semantically rationalised LISP is a possible formalism. The design of such a calculus will be taken up in the next chapter.

3.f.iii. Lessons and Observations

Before leaving the subject of these semantical formulations, as mentioned in the introduction to this section, there are two additional comments to be made that will turn out to be important when we design 3-LISP. The first has to do with passing the structural field as an explicit argument among the meta-theoretic interpretation functions. This field argument is referenced, of course, by applications formed in terms of seven primitive procedures: CAR, CDR, PROP, CONS, RPLACA, RPLACD, and PUT-PROP. The first three use the argument to determine CAR, CDR, and property-list relationships; the last four modify the field in ways that subsequent computations can see.

We can make a very strong claim about the use of the field as an argument: *there is no structure, beyond simple sequencing, to the interactions among calls to these functions.* Though the field itself *has* structure, the terms in the meta-linguistic characterisations of the semantics of 1-LISP procedures always pass the field to their arguments that they receive from their caller, with the exception of modifications (for example by RPLACA) that then hold for the indefinite future. No reference is ever made by a meta-theoretic variable to a field other than the one currently passed around. This is very different from the environment and continuations: continuations often embed other continuations within them, and after processing a procedural reduction (our new term for a procedure application) the environment in force *before* the reduction is again used, whereas *during* the reduction a different environment (the one in force when the closure was created) is used to process the procedural body.

This fact is of course predictable, if one thinks just a moment about what the structural field is. The meta-theoretic mathematical entity is merely designed to *model* an actual structural field, which is a single graph of symbolic structures examined and manipulated during the course of a computation, as sketched in what we called a *process reduction* model of computation in chapter 1. The structural field is not itself part of the semantic domain D — this was what we meant in chapter 2 when we said that the primitive relationships that constitute it are not objectifiable within LISP. Therefore there is no way in which past states of that field can be retained in structures, or designated by symbols in the field. The field is the world in which the processor lives, a world from which that

processor cannot escape, and a world it cannot in one glance see.

The *mathematical* consequence of this fact is this: a single so-called *global* variable could be used, rather than an explicit argument passed around between functions. However there is a deeper way of making the same point. The field of structures is the world over which the programs run: they can *affect* it, but they cannot create or store whole fields. It is this realisation that led us to a characterisation of computation as the interaction between a *processor with state* and a *field with state*; the two are *independent* exactly because there is no sense in which the state of the field is *part* of the state of the processor.

So long as we are using the λ-calculus as our meta-language, we cannot make formal use of this fact, since we have no such mechanism as global variables. However when we embed the meta-theoretic characterisation of LISP into the 3-LISP reflective processor, we will not need to have a structural field *argument* for any of the meta-theoretic procedures: the field is simply there, accessible to and modifiable by any program that wants to touch it. It is the blackboard on which the processor reads and writes: there is no need — no sense in the suggestion — that the blackboard *itself* needs to be encoded on the blackboard. The *environment*, however, will be designated by a particular structure, and passed around as an argument, because each reflective level will operate in a distinct environment. It is for this reason that throughout our analysis we have maintained these two parts of the context in different theoretical objects.

The second comment about our approach has to do with the context arguments to Ψ and Γ. Those functions — the semantical functions explicating procedural consequence — are in some sense odd, since they do not deal with what symbols designate, and yet at the same time they do not tell the formal story of how the computation is effected. Nonetheless we were able to formalise them, and show how they made sense of a variety of phenomena in need of explanation. One role of particular relevance to us was that they are the functions designated by the meta-circular processor.

We commented in this regard that Ψ was crucially a function from $\mathbf{s}$ to $\mathbf{s}$ — from structures onto structures. Its context arguments, however — environments and continuations — were abstract functions; they were not encoded as structural field fragments. Thus, although Ψ and Γ deal with internal procedural consequence, they are

formulated in terms of abstract theoretical posits, not in terms of structurally encoded representations of computational state.

This is both correct and important. It is correct because, as we have maintained all along, the context is borne by these entities *only as functional theoretical posits*: they do not have a reality in the phenomenal world being explained prior to the articulation of the theory explaining it. Rather, they are reified in service of constructing an adequate and finite theory of the wide potential behaviour that forms the subject matter of the theory. More formally, terms in the theory of LISP *mention* LISP s-expressions, and *mention* the functions and numbers that they designate. Terms signifying continuations and environments, however, are *used* in formulating these explanations. For example, in a meta-theoretic expression like

$$\forall S \in \mathit{S},\ E \in \mathit{ENVS},\ F \in \mathit{FIELDS} \quad [G(S,E,F)] \qquad \text{(S3-201)}$$

the term S designates an s-expression, and the term E designates an environment. No s-expression designating or encoding an environment is needed, nor is any mentioned. It is for this reason that, although the notion of an environment plays a crucial role in explaining how LISP works, no ENVIRONMENT-P predicate can be defined within LISP:

> *Environments and continuations are not part of 1-LISP's semantical domain D. They exist only in the semantical domain of the meta-language used to characterise 1-LISP.*

This issue is important because it arises in the design of 3-LISP, when we reify abstract descriptions of processors with structural field expressions. The question we will have to face is this: when a process reflects, and binds formal parameters (ENV and CONT, say) to the environment and continuation in force at the point in the computation prior to reflection, should ENV and CONT designate environments and continuations, or should they designate *structures* encoding environments and continuations. The foregoing argument shows how the only correct answer is the former. It takes another intensional act to access environment and continuation designating terms.

The point is perhaps best stated as follows. At the object level, s-expressions are explicitly used; environments and continuations are *tacit* — part of the background in which those s-expressions are used. One level of reflection moves the s-expressions that were used into a position whereby they are now mentioned; *it simultaneously moves what*

was tacit into terms that are used. Thus at the first reflective level we *use* vocabulary to refer to what was tacit at the object level; we *mention* the terms that were *used* at the object level. Only at the second level do the terms, introduced at the first level to refer to the tacit structure of the object level, become available themselves to be mentioned.

This identification of the reflective hierarchy with an increasingly reified account of the tacit structure of non-reflective behaviour is one of the most striking aspects of our design of 3-LISP, and it will receive much more treatment in chapter 5. What is notable at this point is how some of those properties have already been embodied in decisions we have made in our mathematical characterisation of our simple initial dialect.

3.f.iv. Declarative Import, Implementation, and Data Abstraction

One postscript remains. The reader will have noticed that we place great emphasis on the apparently static structure of the entities in the structural field — what might seem an odd emphasis in light of the current interest in *data abstraction.* In particular, it may seem as if we are putting theoretical weight on what is normally considered part of the *implementation,* where only the resultant *behaviour* is what counts.

There are several replies to be made to this apparent criticism. First, we have taken some pains to define the structural field abstractly, and not to let our characterisation of *it* be influenced by matters of implementation — by considerations, in particular, of how it might be encoded in the structural field of an implementing process. For example, in defining the LISP field we did not mention the notion of a *pointer,* a type of object almost universally used to implement the LISP field in the memory of a Von Neuman underlying architecture. Thus we are focusing on the structural field *of an abstract or virtual machine*; there is no limit to how abstract a structural field one could examine in this way. So the question reduces to one, not of *implementation,* but of the legitimacy of focusing on a structural field at all.

With respect to this question, it was our claim, in sketching the process reduction model of computation in the first chapter, that the notion of a field of structure in fact permeates a great many calculi, *because of the fact that we attribute declarative import to computational structures.* Furthermore, we include (in LISP's case) all programs in the structural field, and all programming languages, even if the programs *engender* behaviour of

one sort or another, are nonetheless static structural objects *qua programs.*

In addition, the knowledge representation hypothesis, under whose influence the present project is pursued, makes a strong claim about the form and organisation of the elements of the structural field. It is exactly the substance of this hypothesis that the most compelling functional decomposition of an intelligent process will posit, as theoretically substantive ingredients in a process, a set of structures on which declarative import can be laid. No mention is made of how abstractly defined these structures will have to be, and it is in order to facilitate very abstract machines that we have defined the notion of a structural field, and distinguished it completely from issues of notation. In other words, if one abandons completely any notion of "static" symbols, and concentrates purely on behaviour, it is indeed possible to deny the utility of the notion of a structural field. The price will be that one would have to deny any representational claims in addition. It is probable as well that one would have to give up any notion of symbol, any notion of language, and probably any recognisable notion of processing.

(There is no doubt, in other words, that viewing computational processes purely behaviourally — and ignoring any semantical claims on their ingredients — is a more general approach. The problem is that it is far *too* general to be of any interest: it may even be too general to count as *computational.*)

In spite of these rejoinders, however, an extremely important issue remains. One of the most compelling aspects of computational systems is the ease with which they allow programmers to define abstract data types out of more primitive ones, in a manner analogous to the way in which procedures are defined in terms of more primitive ones. A standard example is the notion of a complex number, which can be easily represented either in terms of its rectangular or polar coordinates. For some purposes one representation is more convenient; for others, the other makes calculations simpler. Suppose for example we choose the first option, representing a complex number as a list of its real and imaginary rectangular components. Thus we might define a complex number as a list; the real and imaginary coordinates being the first and second elements. Thus, if c_1 is a complex number, `(CAR` c_1`)` would designate the real component, and `(CADR` c_1`)` would designate the imaginary component (notice we say *designate*, not *return* — this by way of preparation for `2-LISP`). In order to obtain the radius, one would use the expression:

```
(SQRT (+ (* (CAR C) (CAR C))                                    (S3-202)
         (* (CADR C) (CADR C))))
```

Similarly the angle could be computed by taking the appropriate arctangent:

```
(ARCTANGENT (CAR C) (CADR C))                                   (S3-203)
```

No one, of course, would use copies of such implementation-dependent code scattered throughout a body of code. It is widely considered more modular, rather than deciding once and for all between these two options, to define what is called an *abstract* data type of "complex number", on which a number of operations are defined. Suppose for example we require that for any complex number C we be able to use the forms (REAL C), (IMAGINARY C), (RADIUS C), and (ANGLE C) to refer, respectively, to the two rectangular components, and to the two polar components. We would define some way of *storing* the information within this module, and would define the procedures appropriately. Of course to make the example realistic we have to provide a way to construct imaginary numbers; we will assume two additional functions: COMPLEX-FROM-RECTANGULAR and COMPLEX-FROM-POLAR that, given two coordinates in the respective system, would construct one instance of the abstract data type, appropriately constrained. For example, the following module yields this behaviour, implementing complex numbers in terms of their rectangular coordinates:

```
(DEFINE REAL                                                    (S3-204)
   (LAMBDA EXPR (C) (CAR C))

(DEFINE IMAGINARY
   (LAMBDA EXPR (C) (CADR C))

(DEFINE RADIUS
   (LAMBDA EXPR (C)
      (SQRT (+ (* (CAR C) (CAR C))
               (* (CADR C) (CADR C))))))

(DEFINE ANGLE
   (LAMBDA EXPR (C) (ARCTANGENT (/ (CADR C) (CAR C)))))

(DEFINE COMPLEX-FROM-RECTANGULAR
   (LAMBDA EXPR (REAL IMAG) (LIST REAL IMAG)))

(DEFINE COMPLEX-FROM-POLAR
   (LAMBDA EXPR (RAD ANG)
      (LIST (* RAD (COSINE ANG))
            (* RAD (SINE ANG)))))
```

Analogously, we could have the dual implementation, in terms of polar coordinates:

```
(DEFINE REAL                                                    (S3-205)
   (LAMBDA EXPR (C) (* (CAR C) (COSINE (CADR C)))))

(DEFINE IMAGINARY
   (LAMBDA EXPR (C) (* (CAR C) (SINE (CADR C)))))

(DEFINE RADIUS
   (LAMBDA EXPR (C) (CAR C))

(DEFINE ANGLE
   (LAMBDA EXPR (C) (CADR C))

(DEFINE COMPLEX-FROM-RECTANGULAR
   (LAMBDA EXPR (REAL IMAG)
      (LIST (SQRT (+ (* REAL REAL) (* IMAG IMAG)))
            (ARCTANGENT (/ IMAG REAL)))))

(DEFINE COMPLEX-FROM-POLAR
   (LAMBDA EXPR (RAD ANG) (LIST RAD ANG)))
```

Outside of these modules only the six procedures names would be used; since the behaviour (modulo efficiency considerations) of the two is the same, external programs need not know which implementation strategy has been used.

It is clear that arbitrary types of object in the user's world can be handled in a like manner: our example is extraordinarily simple, but it is not uncommon to define, in this same style, abstract types to represent objects as complex as files, display-oriented input/output devices, and so on. The question for us — the reason that these considerations matter in our investigation — has to do with how to characterise such computational structures semantically. From a *procedural* point of view the standard techniques will suffice, although it requires some effort to make these abstractions clear in the semantical treatment — to make their borders, in other words, come to the fore in the mathematical characterisations that emerge. But what is much less clear is how to make the *declarative* import of such a computational module explicit. *How do we say*, for instance, with respect to the example we gave above, *that it represents a complex number?* How would we say of a far more complex artifact that it (or instances of it) designate graphical terminals? To what extent, in other words, are the notions of declarative import and data abstraction related?

There are a variety of hints that may be taken from a close examination both of what we actually did in the example above, and from a consideration of the terminology that is typically used to describe such abstractions. First, in spite of the received maxim

that behaviour is what is crucial, in writing down the code in S3-204 and S3-205 — the code that is intended to *generate* that behaviour — we did not in some magic fashion build it out of behaviour; as we always do, we wrote down static symbols, the processing of which is intended to yield the behaviour we had in mind to implement. It follows, therefore, that the code we used itself must succumb to a declarative treatment, based on whatever interpretation function was in effect prior to the definition of complex numbers. It is entirely likely that this characterisation will be at odds with the one we are headed for — there is no likelihood whatsoever that Φ of the structures given above will have anything to do with instances of complex numbers — but it is not too much to ask that we establish some sort of *relationship* between the semantical account that emerges from the code we have written, and the semantical account, in terms of complex numbers, that we wish to explicate.

Furthermore, as well as this *code* having determinable semantical import, any given instance of the abstract data type will necessarily have some implementation in terms of elements of the structural field of the implementing machine. That structural field fragment will itself have declarative import, as described by the standard semantics. We can in fact readily determine the declarative import of such instances in our simple example. First, however, we need to clarify our terminology. Our new data type is not really that of a complex *number*; rather, we will call the data type a ***complex numeral***, since really what we have done is implement an abstract formal object to which we intend to attribute the following semantical import: a complex numeral will designate a complex number.

Then, since all expressions of the form (LIST X Y) designate the two-element sequence of the referents of X and Y, it is clear that on either implementation, a complex numeral C will be taken by our semantics onto a sequence of two numbers. Actual complex numbers, of course, are precisely *not* a sequence of two real numbers. Rather, and this is what we know when we accept the implementation, the information about a *particular* complex number can be deduced from the following two things: a *general* claim about a bijection between complex numbers and two real numbers, and two *particular* real numbers that represent the given complex number.

Suppose we define a relationship Π that encodes the appropriate mapping between sequences of real numbers and complex numbers (we will focus on the rectangular implementation, although the form of the argument is identical in either case). Thus for example Π(2,3) = 2 + 3i. The crucial fact about Π is that it be formulated in terms of the *designation*, in the standard semantical treatment, of the implementation of complex numerals. In other words, if c_1 is a complex numeral — a two-element list of real numerals — returned by COMPLEX-FROM-POLAR, then Π(ΦEF(c_1)) is the complex number that c designates in what we are beginning to think of as an extended calculus.

Once we had defined Π, we would have to specify the consequences, in its terms, of the significance of the abstract operators defined over the data type. For example, we would want to prove that the function designated by REAL was (the extensionalisation of) a function from complex numbers to their real coordinates. Suppose that REAL-OF and IMAGINARY-OF are two functions in out meta-language that project complex numbers onto their real and imaginary coordinates. In other words we are assuming that:

Π(C) = REAL-OF(C) + [IMAGINARY-OF(C)]i (S3-206)

Then what we would want to prove would be something like the following:

ΦEF("*REAL*) = EXT(λX . REAL-OF(Π(X))) (S3-207)

Similarly for all of the various other functions comprising the behaviour defined over complex numbers.

Now if this is done, some remarkable properties emerge. First, suppose we define an extended semantical interpretation function Φ', which is intuitively just like Φ except it is extended to include Π. In other words, if Φ of a term is in the domain of Π, then Φ'(X) = Π(Φ(X)); otherwise Φ'(X) = Φ(X) (this would of course be contextually relativised as usual). Then what is true is that 2-LISP (or whatever rationalised dialect one uses) *would be Φ'-preserving as well as Φ-preserving.* For if the primitive language processor preserves Φ-designation, and if the *implementation* relationship is defined over *referents*, not over *structures*, then it is obvious that a regime that maps one term into another with the same referent will not change any properties that depend only on reference.

Furthermore, if function application is redefined to use Φ instead of Φ, then such equations as S3-207 could be written as follows:

```
Φ'EF("REAL) = EXT(λX . REAL-OF(X))                    (S3-208)
```

In other words a systematic way emerges in which the interpretation functions can be extended along with the introduction of new abstract data types, so that the fundamental semantical characteristics of the underlying system are preserved.

In other words, if a user simply *posits* the designation of code implementing abstract data — simply asserts, for example, that REAL designates the real coordinate of a complex number, without proving it or relating it to the semantics of the implementing language — then nothing about the semantical properties of the processing of this instances of this data type can be said, and not surprisingly. If, however, such abstract data type extensions can be proved as sound and consistent, in terms of the designations of the implementing programs, then the semantical soundness — and, for example, the semantical flatness of the underlying processor — carry over from the implementing language onto the language extended with the abstract data type. The moral, in other words, is that if the abstract data type is soundly defined and implemented *in terms of the semantical import of a semantically rationalised dialect*, then the resultant *extended* dialect will be semantically rationalised as well. This is quite considerable a result, for it means if we define 2-LISP correctly, even if it is a simple kernel calculus, nonetheless we (or any other user) will be able to build it up in standard powerful ways. If that extension is done with care, then its underlying semantical cleanliness will perfuse the abstract structures implemented on top of it.

Chapter 4. 2-LISP: A Rationalised Dialect

We turn next to the design of 2-LISP, a dialect *semantically* and *categorically* rationalised in terms of the analysis set forth in the previous chapter. The most striking property of 2-LISP that differentiates it from 1-LISP is of course the fact that its procedural regimen is based on a concept of *normalisation* rather than of *evaluation* — with the concomitant commitment to a declarative semantics defined prior to, and independently of, procedural consequence. We will attempt to show, in keeping with this approach, that a clear separation between the *simplification* and *reference* of expressions can workably underwrite the design of a complete and practicable system (something that no amount of abstract argument can demonstrate). In addition, there are two further points that 2-LISP is intended to demonstrate, emerging from our drive to free the meta-structural powers of a computational calculus for reflective purposes. In particular, we observed in chapter 2 that the 1-LISP meta-structural facilities were employed for the following reasons (among others):

1. To partially compensate for the lack of higher-order functionality in a first-order system.
2. To deal with certain forms of objectification and compositionality of program structure in the structural field.

The SCHEME language has shown us that a LISP need not use meta-structural capabilities to deal with higher-order functionality, but even in that dialect certain types of objectifications required meta-structural treatment (the explicit use of EVAL and APPLY). We saw as well that the objectification issue was not treated in the λ-calculus; currying, the standard way in which multiple arguments are handled, provides no solution. We will show in 2-LISP that both facilities -- higher-order functionality and the ability to objectify multiple arguments — can be conveniently and compatibly provided in a semantically-rationalised base language. Meta-structural primitives, in other words, are necessary for neither capability.

It does not follow that 2-LISP will have no meta-structural primitives: on the contrary, simple naming and de-referencing primitives will be introduced and rather thoroughly examined. In addition, we will initially provide primitive access to 2-LISP's main processor functions (under the names NORMALISE and REDUCE). Strikingly, however, we

will be able to *prove that there is no reason one would ever need to use them* — or, to put the same point another way, *we will show that they need not be primitive, but could be defined in terms of other functions* (this is a claim called the *up-down* theorem, proved in section 4.d.iv). This is a much stronger result than we were able to reach in 1-LISP or SCHEME, and it is just the right preparation for 3-LISP, where these functions will be re-introduced, as part of the reflective capability, and used in defining programs that are simply not possible in 2-LISP (those that objectify the state of the processor in the midst of a computation). The point is that NORMALISE and REDUCE will be required only when the processor state (in terms of an environment and continuation) must be objectified: in other cases, less powerful primitives will always suffice.

In the course of our pursuit of these goals, we are also committed to two aesthetic principles:

1. In all aspects of the design the *category* (as opposed to *individual*) identity of a form should determine its significance (i.e., there should be no distinguished individuals that receive special treatment).
2. To the maximum extent possible, there should be category alignment across the entire system: among lexical notation, structural field, declarative import, and procedural consequence.

There are several properties of 2-LISP that should be made clear at the outset. First, 2-LISP is an extremely powerful calculus in various formal senses: it will handle functions of arbitrary order; it contains primitive intensional operators, both functional (LAMBDA) and hyper-intensional (QUOTE and primitive support for arbitrary IMPRS); it contains powerful meta-structural facilities; and it provides primitive access to the main processor function. It is our claim that these facilities can all be provided in a clean manner, but there are of course consequences to this power, such as that it will in general be undecidable what function a given 2-LISP program computes, and so forth.

In spite of this power, however, there is an odd sense in which 2-LISP is not very well self-contained — it does not provide a very natural closure of capabilities over the concepts in terms of which it is defined. In particular, various facilities of 2-LISP lead the programmer into odd behaviours and curious problems, some of which have no obvious solution. For example, we will show how IMPRS (intensional procedures that do not normalise their arguments) have no way of normalising those arguments after the fact, since the appropriate context of use has been lost by the time the body of the intensional

procedure is processed. Thus for example if we were to write

```
(LET [[X 3]] (TEST X))                                    (S4-1)
```

in 2-LISP, and if TEST signified an intensional procedure, there would be no way within the body of TEST to ascertain that X was bound to 3 at the point of call. The problem — which we will explore in greater detail in section 4.d — arises from the interaction between the meta-structural nature of IMPRS and the static scoping protocols the govern 2-LISP variable binding.

Similarly, in setting out the structure of 2-LISP closures (normal form function designators), we will be forced to accept encodings of environments as structural constituents. It was part of our stated goal in designing 2-LISP, however, to *avoid* the introduction of structural encodings of theory-relative meta-theoretic posits. Environments were to be entities in the *semantic domain of the meta-theory* that facilitated our explanation of how 2-LISP worked; we intended to postpone introducing environments *into LISP itself* until we took up reflection as an explicit concern. That pristine goal, however, will elude us, because (as we will show) we still lack an appropriate theory of (finitely representable) functions-in-intension. As a consequence we will be forced to use environment encodings as a stop-gap measure to cover for this lack.

Many other examples could be cited, but they will arise in due course: this is not the place to pursue details. The general character of these problems, however, worth noting at the outset, is that we will find no solutions, nor even any hint that solutions are possible. Nor do other systems provide any clues: since the λ-calculus has no meta-structural facilities, the questions do not arise in its case, and it is striking that SCHEME does not provide EVAL and APPLY as primitive procedures, perhaps for some of these very reasons.

At heart, the problem — with IMPRS and closures and all the rest — is that they inevitably force us to take *part* of a step towards full reflection, without taking the *whole* step. In 3-LISP, a *reflective procedure* (a category that will subsume IMPRS) will enable us at will to bind not only designators of arbitrary argument expressions, but also fully informative designators of arbitrary contexts. The ability to objectify the environment in this way doesn't so much *require* reflection — it would be more accurate to say that it *is* reflection. The present moral is that the full complement of natural 2-LISP facilities cannot be developed without such a capability: that, in a word, 2-LISP is inherently incomplete.

One could of course argue that these results suggest that 2-LISP is already too powerful — that we should restrict it so as not provide *any* meta-structural powers. This makes a little sense, since the fact that 2-LISP can handle objectification and higher-order functionality at the base level means that many of the standard LISP reasons for wanting meta-structural powers are obviated. On the other hand, there remain cases — programs that write programs are an obvious example — where such powers are essential. One could argue instead that although meta-structural powers over essentially uninterpreted expressions may be useful, we could perhaps avoid mentioning structures that were actual parts of 2-LISP programs. But the closure question (the encoding of environments in closures) arose simply in providing an adequate treatment of higher-order functionality.

We will leave meta-structural facilities in 2-LISP, but we will not attempt to find a natural boundary for them, since this author, at least, does not believe any such suitable limits can be found. Rather, we consider 2-LISP a step on the way towards the yet more powerful 3-LISP, which does provide a natural closure of meta-structural powers. The intents of this chapter, in other words, are two: first, to demonstrate the power and effectiveness of our double semantics viewpoint; and second, to make evident the fact that a procedurally reflective dialect is not an esoteric dream, but merely the natural reconstruction of current practice. A good hard look at 2-LISP, in fact, not only pushes us irretrievably towards 3-LISP; it almost dictates the structure of that further dialect. 2-LISP, in sum, is a stepping stone; 3-LISP will be the final product.

4.a. The 2-LISP Structural Field

In this first section we will present the 2-LISP structural field — and, as it happens, the 3-LISP field, since the latter dialect is structurally identical to the former, differing only in having a processor with extended power. We will work primarily with six categories, in both syntactic and semantic domains; 2-LISP will be approximately constituted as follows (the notational BNF is a little informal: a more accurate version would introduce breaks between identifiers, and so forth, but the intent should be clear):

SF Category	*Designation* (Φ)	*Notation* (ΘL) (S4-2)
Numerals	Numbers	`[ "+" | "-" ]° digit [digit]*`
Booleans	Truth-values	`[ "$T" \| "$F" ]`
Rails	Sequences (of Φ's of elements)	`"[" [formula]* "]"`
Pairs	Functions, and values of applications	`"(" formula "." formula ")"`
Handles	S-expressions	`"'" <notation of referent>`
Atoms	Φ of bindings, and user's world	`non-digit [character]*`

The first four semantic types (numbers, truth-values, sequences, and functions) are mathematical and abstract, the fifth is the structural field itself, and the sixth is whatever extension is required in a particular use of a 2-LISP program. It is not coincidental that there are six primary structural categories and six primary semantical categories — we will be able to set these two taxonomies into approximate correspondence, as discussed in the previous chapter, and as is suggested in the table just presented. The pairing cannot be exact, however, in part because pairs — encodings of function applications — can of course designate any element in the semantical domain, as can atoms (names).

4.a.i. Numerals and Numbers

As in 1-LISP, the 2-LISP field contains an infinite number of distinct numerals corresponding one-to-one with the integers. Each numeral is atomic, in the sense that no first-order relationships are defined as functions over them; in addition, no other elements of the field are accessible from the numerals (other than their handles: see section 4.a.vi.). They are notated in the standard fashion, as explained in chapter 2. Furthermore, each numeral will designate its corresponding integer in all contexts. Using the machinery of the last chapter, we can summarise these points (the function M in S4-7 is the standard interpretation function from numerals to numbers; S is the set of structural field elements):

INTEGERS ≡ { I | I is an integer} (S4-3)

NUMERALS ≡ {N | N ∈ *S* ∧ N is a numeral } (S4-4)

L-numeral ::= ["+"_|"-"_]° digit_[digit]* (S4-5)

Θ_L(L | L is an L-numeral) ∈ ***NUMERALS*** in the standard fashion (S4-6)

∀E ∈ ***ENVS***, F ∈ ***FIELDS***, N ∈ ***NUMERALS*** [ΦEF(N) = M(N)] (S4-7)

Equation S4-7 implies that each numeral designates an integer; that this designation is one-to-one is implicit in S4-5 and S4-6; thus the following is provable:

∀I ∈ ***INTEGERS*** ∃N ∈ ***NUMERALS*** (S4-8)
[∀E ∈ ***ENVS***, F ∈ ***FIELDS***
[[ΦEF(N) = I] ∧
[∀M ∈ ***NUMERALS*** [[ΦEF(M) = I] ⊃ [M = N]]]]]

Numerals will be taken as *canonical normal-form designators of numbers*: thus any 2-LISP structure S that designates a number (and that normalises at all) must normalise to the numeral that designates that number. Thus we have our first constraint on 2-LISP's Ψ (it should be clear that so far this behaviour is no different from that of 1-LISP):

∀E ∈ ***ENVS***, F ∈ ***FIELDS***, S_1, S_2 ∈ *S* (S4-9)
[[[ΦEF(S_1) ∈ ***INTEGERS***] ∧ [S_2 = ΨEF(S_1)]] ⊃
[S_2 = M^{-1}(ΦEF(S_1))]]

It should be clear, however, that S4-9 is a *desideratum* that we will want to prove: we cannot simple postulate it, since it does not yield an algorithmic method by which it may be rendered true. Rather, we will start simply, with the fact that numerals normalise to themselves:

∀E ∈ ***ENVS***, F ∈ ***FIELDS***, N ∈ ***NUMERALS*** [ΨEF(N) = (N)] (S4-10)

Finally, the normalisation of numerals involves no side effects, as is indicated by the following characterisation in terms of total procedural consequence.

∀E ∈ ***ENVS***, F ∈ ***FIELDS***, N ∈ ***NUMERALS***, C ∈ ***CONTS*** (S4-11)
[Σ(N, E, F, C) = C(N, M(N), E, F)]

From S4-7 and S4-10 it follows that numerals are *context-independent*, from S4-10 it follows as well that they are *stable*, and from S4-11 it follows that they are *side-effect free*. Thus we straight away have shown the truth of the following:

∀N ∈ ***NUMERALS*** [NORMAL-FORM(N)] (S4-12)

We cannot yet show very many examples, since we have introduced so little, but at least the following follows from what has been said (we use the symbol "⇒" to indicate the lexicalisation of the *normalisation* relationship — i.e., the relationship between two lexical notations, where the second notates the result of normalising that structure notated by the first — just as we used "→" to indicate the lexicalisation of evaluation):

```
4          ⇒   4                                   (S4-13)
-26        ⇒   -26
00000111   ⇒   111
-0         ⇒   0
```

4.a.ii. Booleans and Truth-Values

There are two 2-LISP boolean constants, comprising their own structural field category, and designating respectively Truth and Falsity. They are like the numerals in several ways: they are atomic; no other structures (besides their handles) are accessible from them; and they are the canonical normal-form designators of their referents. We will not use the name NIL to notate the boolean that designates Falsity, but a distinguished element used for no other purpose. As hinted in S4-2, we will instead notate them not simply as "T" and "F", but as "$T" and "$F", in order to distinguish the booleans lexically (from the atoms), as well as structurally and semantically ("$" is otherwise a reserved letter in 2-LISP notation). The inconvenience in requiring an extra letter is more than compensated for by the maintenance of the category alignment.

The equations constraining the booleans are similar to those describing the numerals. First we have the equations defining the form and designation of the booleans:

TRUTH-VALUES ≡ { Truth, Falsity } (S4-14)

BOOLEANS ≡ {"$T, "$F} (S4-15)

L-boolean ::= ["$T" | "$F"] (S4-16)

Θ_L("$T") = "$T Θ_L("$F") = "$F (S4-17)

∀E ∈ ***ENVS***, F ∈ ***FIELDS***, B ∈ ***BOOLEANS*** [ΦEF(B) = TRUTH-VALUE(B)] (S4-18)

The constraint we will ultimately want to prove is that all expressions that designate truth or falsity (all ***sentences***, to use a definition from logic) and normalise at all, normalise to

the appropriate boolean constant:

$$\forall E \in \textit{ENVS},\ F \in \textit{FIELDS},\ S_1,\ S_2 \in \textit{S} \qquad \text{(S4-19)}$$
$$[[[[\Phi EF(S_1) = \text{Truth}] \wedge [S_2 = \Psi EF(S_1)]] \supset [S_2 = \text{"\$T"}]] \wedge$$
$$[[[\Phi EF(S_1) = \text{Falsity}] \wedge [S_2 = \Psi EF(S_1)]] \supset [S_2 = \text{"\$F"}]]]$$

Again, we can posit this as true of the booleans themselves, and can also assert that these two constants are side-effect free:

$$\forall E \in \textit{ENVS},\ F \in \textit{FIELDS},\ B \in \textit{BOOLEANS}\ [\Psi EF(B) = B] \qquad \text{(S4-20)}$$

$$\forall E \in \textit{ENVS},\ F \in \textit{FIELDS},\ B \in \textit{BOOLEANS},\ C \in \textit{CONTS} \qquad \text{(S4-21)}$$
$$[\Sigma(B,E,F,C) = C(B,\text{TRUTH-VALUE}(B),E,F)]$$

Thus, as was the case with the numerals, we have shown that the booleans satisfy the normal-form constraint (S4-18, S4-20, and S4-21):

$$\forall B \in \textit{BOOLEANS}\ [\text{NORMAL-FORM}(B)] \qquad \text{(S4-22)}$$

Again, only the most simplistic of illustrations are possible:

$$\text{\$T} \Rightarrow \text{\$T} \qquad \text{(S4-23)}$$
$$\text{\$F} \Rightarrow \text{\$F}$$

4.a.iii. Atoms

Like the numerals and booleans, 2-LISP atoms are structurally similar to those of 1-LISP. They are atomic and indivisible, and there are assumed to be an infinite number of them in the field. Each is notated with a lexical type in the usual way, with distinct lexical *types* (except with respect to the case of the constituent characters) notating distinct individual atoms. Again, *in the field* only their handles are accessible: we will discuss environments presently.

$$\textit{ATOMS} \equiv \{ A \mid A \text{ is an atom} \} \qquad \text{(S4-24)}$$

$$\text{L-atom ::= [character_]* non-digit [_character]*} \qquad \text{(S4-25)}$$

$$\Theta_L(L \mid L \text{ is an L-atom}) = \text{the corresponding atom} \in \textit{ATOMS} \qquad \text{(S4-26)}$$

For the time being we will not define a property list relation as a function over atoms — although such an extension would need to be explored for a practical version of the dialect.

Semantically, all atoms will be viewed as *context-dependent names*, in the sense that all atoms will designate the referents of their bindings in the appropriate environment, and

they will also normalise to those bindings. *No atoms*, in other words, *will be viewed as constants*, and, correspondingly, *no atoms are in normal-form.* It follows that no atoms, including the names of the primitive procedures, will normalise to themselves: rather, atoms must normalise to normal-form designators of the referents of their bindings. Finally, as will be discussed in section 4.b, the primitive procedures are not defined in terms of atoms, but rather in terms of primitively recognised closures. The first of these points is easily stated:

$$\forall E \in \textit{ENVS},\ F \in \textit{FIELDS},\ A \in \textit{ATOMS}\ [\Phi EF(A) = \Phi EF(E(A))] \qquad \text{(S4-27)}$$

$$\forall E \in \textit{ENVS},\ F \in \textit{FIELDS},\ A \in \textit{ATOMS}\ [\Psi EF(A) = E(A)] \qquad \text{(S4-28)}$$

These two equations, however, do not imply that no atoms are in normal-form, since we have yet to identify how environments can be affected. It will turn out to be a theorem about 2-LISP that all *bindings* are in normal-form, but that will have to be proved, and follows from the way in which reductions are treated, as shown below.

The normalisation of atoms is also side effect free:

$$\forall E \in \textit{ENVS},\ F \in \textit{FIELDS},\ A \in \textit{ATOMS},\ C \in \textit{CONTS} \qquad \text{(S4-29)}$$
$$[\Sigma(A, E, F, C) = C(E(A), \Phi EF(E(A)), E, F)]$$

Examples of the normalisation of atoms will be given once we have some machinery for building environments. It should be noted as well that we will use the term "atom" when we refer to these objects from a primarily structural, non-semantic point of view. Functionally, atoms play a role as context relative names; when we wish to emphasise their use rather than their structure, we will variously call them ***variables*** or ***parameters***.

4.a.iv. Pairs and Reductions

Although 2-LISP pairs are identical to 1-LISP pairs from a purely *structural* point of view, some substantial differences between 2-LISP and 1-LISP will begin to emerge between the dialects as we look at their semantics, procedural treatment, and notation. In particular, we assume an infinite number of distinct pairs, over which the standard two first-order asymetric relationships are defined, called CAR and CDR. These relationships are total functions, mapping each pair onto some arbitrary element of the 2-LISP field. The primitive notation for pairs is like that of 1-LISP (with all its problems): a pair is notated in terms of the notations of its CAR and CDR, enclosed within parentheses and separated by a

dot, and every *reading* of a lexical combination notates a *previously inaccessible* pair:

```
PAIRS ≡ { P | P is a pair }                                        (S4-30)
CARS  ≡ [ PAIRS → S ]
CDRS  ≡ [ PAIRS → S ]

L-pair ::= "("_formula_"."_formula_")"                             (S4-31)

Θ_L(L | L is an L-pair) = a pair ∈ PAIRS whose CAR is Θ_L of the    (S4-32)
     1st formula and whose CDR is Θ_L of the 2nd
```

It would be possible to define a radically different kind of lexical notation for pairs, with fewer ambiguities, less incompleteness, and so forth, but such a move is major change, especially since it is only through notation that we humans access the LISP field, and therefore preserving LISP's notational style is part, if not all, of our claim to still be defining a dialect within the LISP family. For these reasons, although we do not endorse the properties this notation brings with it, we will stay with tradition. We will not, however, define the usual notational abbreviation for lists (since we are not defining lists at all in 2-LISP), but will instead reserve that notational style for a combination of pairs and rails, as shown below.

Semantically, pairs will be taken to designate the value (note our use of the term "value") of the function designated by the CAR applied to the CDR (not, one may note, to the object designated by the CDR). Two facts make this different from 1-LISP. First, in 1-LISP we defined the declarative import of a pair as follows (ignoring side-effects for a moment):

```
∀E ∈ ENVS, F ∈ FIELDS, P ∈ PAIRS                                   (S4-33)
    [ΦEF(P) = [ (ΦEF(S1))EF ] <S2 S3 ... Sk>]
        where P = ⌜"(S1 S2 S3 ... Sk)⌝ in F
```

It was this characterisation that made reference to the notion of a list. Our characterisation of declarative semantics for 2-LISP pairs, in contrast, is the following:

```
∀E ∈ ENVS, F ∈ FIELDS, P ∈ PAIRS                                   (S4-34)
    [ΦEF(P) = [ (ΦEF(P1))EF ] P2]
        where P = ⌜"(P1 . P2)⌝ in F
```

According to the meta-language, in other words, all functions designated by 2-LISP expressions are functions *of a single argument.* This will prove simpler in a number of ways, partly because it provides the correct ingredients for our successful treatment of argument objectification within the base language. Note, furthermore, that this is not a case of currying the LISP, in the way that we have curried the meta-language.

Furthermore, it apparently has no adverse consequences, even from the point of view of implementation, as we will ultimately demonstrate. Finally, the 2-LISP characterisation is *total*, in a certain sense, in that it makes reference only to the CAR and CDR of a pair, and, as we have just mentioned, all pairs have CARs and CDRs. There is no way, in other words, for 2-LISP pairs to be structurally ill-formed from a declarative point of view (or from a procedural point of view, as we will show in a moment).

Procedurally (again temporarily ignoring side-effects for pedagogical simplicity), we have a corresponding characterisation: the normal-form of the CAR of a pair is reduced with the CDR according to the function engendered by the internalisation of the CAR's formal form:

$$\forall E \in \textit{ENVS},\ F \in \textit{FIELDS},\ P \in \textit{PAIRS} \qquad \text{(S4-36)}$$
$$[\Psi EF(P) = [(\Delta[\Psi EF(P_1)])EF]\ P_2]$$
$$\textit{where}\ P = \ulcorner(\underline{P_1}\ .\ \underline{P_2})\urcorner\ \textit{in}\ F$$

Again, this should be compared with S3-135. Since no induction is required to identify the arguments, S4-34 and S4-35 can more accurately (in the sense of using F explicitly) be written as follows:

$$\forall E \in \textit{ENVS},\ F \in \textit{FIELDS},\ P \in \textit{PAIRS} \qquad \text{(S4-36)}$$
$$[[\Phi EF(P) = [\ (\Phi EF(F^1(P)))EF\ ]\ F^2(P)\ ] \wedge$$
$$[\Psi EF(P) = [(\Delta[\Psi EF(F^1(P))])EF]\ F^2(P)\ ]]$$

It is to be noted that procedural consequence is defined compositionally; it should also be *true* (if 2-LISP is *correct*) that the following equation holds, but this is a statement we will have to *prove*, from the defining equations such as S4-34 and S4-35, and from the definitions (including the internalisations) of all the primitive 2-LISP procedures:

$$\forall E \in \textit{ENVS},\ F \in \textit{FIELDS},\ P \in \textit{PAIRS} \qquad \text{(S4-37)}$$
$$[[\Phi EF(\Psi EF(P)) = \Phi EF(P)\ ] \wedge \text{NORMAL-FORM}(\Psi EF(P))\ ]$$

In discussing atoms we distinguished between the purely structural term "atom" and the functional terms "variable" and "parameter". Regarding pairs we have a similar distinction: we will use the simple term "pair" again primarily structurally, but will use the term ***redex*** (short for "reducible expression") when more functional or semantic stance is indicated. (There is actually a slight distinction even in reference between the two terms: a pair has only two "parts": a CAR and a CDR; by a redex, however, we will refer to the entire structure involved in a given procedure application — thus the identity conditions are

somewhat different. The details will be spelled out in section 4.a.ix.)

Two things should be noted about how we are proceeding. First, we are simply assuming that we no longer have the kind of troubles with evaluation that prevented us from giving a 1-LISP characterisation of the sort illustrated in S4-34 and S4-35, which allowed as to posit, for example, that $[\Phi E_0 F_0(\text{"+}) = \text{EXT(+)}]$. We will in fact posit just such a declarative import for the symbol "+" in the initial environment, which will work correctly with this semantic charactersation of pairs.

Secondly, the equations we have given of course illustrate the default case only; they do not handle side effects. More properly, therefore, we have to give the full Σ-characterisation of the semantics of pairs. This is given in S4-38 below; note its similarity to S3-135:

$$\begin{aligned}
&\forall E \in \textit{ENVS},\ F \in \textit{FIELDS},\ C \in \textit{CONTS},\ P \in \textit{PAIRS} \qquad \text{(S4-38)}\\
&\quad \Sigma(P,E,F,C) = \Sigma(F^1(P),\\
&\qquad\qquad E,\\
&\qquad\qquad F,\\
&\qquad\qquad [\lambda\langle S_1,D_1,E_1,F_1\rangle .\\
&\qquad\qquad\quad [(\Delta S_1)(F_1^2(P),\\
&\qquad\qquad\qquad\quad E_1,\\
&\qquad\qquad\qquad\quad F_1,\\
&\qquad\qquad\qquad\quad [\lambda\langle S_2,E_2,F_2\rangle .\\
&\qquad\qquad\qquad\qquad C(S_2,[D_1(F_1^2(P),E_1,F_1)],E_2,F_2)])]])
\end{aligned}$$

The importance of this equation, and the role it will play in establishing our main theorems, will emerge later in the chapter.

A number of preparations are still required before we can give examples of the normalisation of pairs. We need, for example, to look at the structural type *rail*, since it is rails that are usually used to encode the arguments to 2-LISP procedures. We also need to show how to designate elements of the field. Finally, of course, we need to define what the primitive 2-LISP procedures are. We will then be in a position to show that, for example, (CAR (CONS 'A 'B)) designates the atom A, as one would expect. Simple examples like this will be given in section 4.b.

4.a.v. Rails and Sequences

In standard LISPs the derived notion of a list, as we remarked, is used both to encode function applications and enumerations (including enumerations of multiple arguments for procedures). We lodged two complaints against this practice: first against the fact that this data structure was not primitive (that fact alone broke the category correspondence between structures and semantics), and secondly against the use of one structure type for two semantic purposes. The foregoing discussion of pairs shows how in 2-LISP we have defined applications, both declaratively and procedurally, directly in terms of pairs, rather than lists. We still lack, however, a structure with which to enumerate a sequence of arguments — or, indeed, a sequence of any entities whatsoever.

There are two related problems coalesced here, in need of clarification. First is a *structural* lack: as posited above, there is no way in which a procedure can be called with more than a single structural argument. The second is a *semantical* inadequacy: we have as yet no accepted way to designate a *sequence of referents.* These, as we should by now expect, are by no means the same problem; we will look at them separately.

With regards to the first, it might seem, at least theoretically, that a possible solution would be to wrap all the arguments to a procedure up into one object before calling it. For example, one could imagine a variant on 1-LISP in which, instead of calling the addition function + with two arguments, one gave it a single list of two arguments, as for example in (+ (LIST 3 4)). But this fails, since the problem recurses: we have no way to define such a LIST function. This is not for lack of an appropriate primitive LIST procedure; the problem is rather that, even if such a procedure existed, there would be no way to call *it* with more than one argument. And if a method were devised by which LIST could be called with multiple arguments, then any function could be called in that way, and LIST would not be needed. Unless, of course, calls to LIST were *structurally* distinguished — but that is too inelegant to contemplate.

Another suggestion would be to employ currying, in the style — typically adopted in the λ-calculus — that we have employed throughout in our meta-language. This is of course possible; the λ-calculus loses no power in virtue of being defined with single arguments. A LISP-like version might encode the addition of 3 and 4 as ((+ 3) 4). This

currying approach has a variety of advantages, and could be made *notationally* equivalent to the old by defining the following as a notational abbreviation:

```
"(" _ formula1 _ formula2 ... _ formulak ")"                    (S4-41)
```

could be taken as an abbreviation for

```
"(...(" _ formula1 _ "." _ formula2 _ ")" _ ... _ "." _ formulak _ ")"    (S4-42)
```

rather than as an abbreviation for the standard

```
"(" _ formula1 _ "." _ "(" _ formula2 _ "." _                   (S4-43)
    _ ... _ "(" _ formulak _ "." _ "NIL" _ ")...))"
```

Similarly, in order to make function definition more straightforward, we would allow the following:

```
(LAMBDA (V1 V2 ... Vk) <BODY>)                                  (S4-44)
```

to be an abbreviation for

```
(LAMBDA V1 (LAMBDA V2 ( ... (LAMBDA Vk <BODY>))))               (S4-45)
```

(Of course S4-44 can't be made to be *structurally* identical to S4-45, since that would contradict S4-42; rather, what we mean is that S4-44, in the new notation, would designate a function of the sort that, in the standard notation, S4-45 would designate. This can be arranged with a suitable definition of LAMBDA.) The structures that resulted would by and large look superficially — which is to say notationally — familiar. For example, given the following procedure definition in the new notation:

```
(DEFINE HYPOTENEUSE                                             (S4-46)
   (LAMBDA (X Y)
      (SQRT (+ (* X X) (* Y Y)))))
```

We would have the following expected result:

```
(HYPOTENEUSE 3 4)    ⇒    5                                     (S4-47)
```

This would work because S4-46 and S4-47 would be notational abbreviations for:

```
(DEFINE HYPOTENEUSE                                             (S4-48)
   (LAMBDA (X)
      (LAMBDA (Y)
         (SQRT . ((+ . ((* . X) . X)) . ((* . Y) . Y))))))
```

and

```
((HYPOTENEUSE . 3) . 4)    ⇒    5                    (S4-49)
```

(Actually S4-48 would be much more complex: we have expanded only the body of the procedure *being defined*; the calls to DEFINE and LAMBDA would similarly have to be uncurried, in order to show the full expansion, but we needn't bother with that here. Note as well that under the new proposal forms with *no* arguments such as (RANDOM) are *notationally* ill-formed: schemes which curry as a method of treating different numbers of arguments will not permit functions to be called with no arguments at all.)

This proposal, however, drives us *further away* from any ability to handle objectified arguments, rather than closer. In particular, suppose we wish to add two numbers, and some term X designates them as a unit. Under the current proposal it is *less* easy than before to engender their addition, rather than *more*; a special procedure would have to be devised that element by element applied the function to the sequence, passing the new derived function along at each step. In addition, if Y were a composite term encoding a function application, and we wished to replace its multiple arguments with a new set (a task of the sort that is liable to arise in reflection), this protocol makes it particularly difficult. Rather than existing as a single list, they have been spread out one by one in a series of explicit redexes. For example, suppose that Y was (+ A B), and we wished to change (actually modify) this to be (+ C D). In 1-LISP this could be effected by (we assume that Y, to use 1-LISP terminology, *evaluates* to (+ A B)):

```
(RPLACD Y '(C D))                                    (S4-50)
```

However in the proposal we are currently considering, the form Y would in fact be:

```
((+ . A) . B)                                        (S4-51)
```

Therefore the modifications would have to be:

```
(BLOCK (RPLACD Y 'D)                                 (S4-52)
       (RPLACD (CAR Y) 'C))
```

And if the *list* (C D) were the value of a single variable, rather than being explicitly decomposed in this fashion, the change would be even more complex.

In sum, while this currying proposal is eminently feasible, since we are working in a higher-order language (the intermediate constituents of a curried application are functions, of course, which is straightforward in a higher order formalism; currying would of course not work in 1-LISP), the currying approach does nothing to answer our original goal.

Furthermore, the currying suggestion doesn't even *address* the second of our original concerns. What is true about the suggestions just considered is that they are purely *structural* suggestions: they do not deal with the related but distinct question of what it is to *designate* an abstract sequence of objects. It follows from our semantics, for example, that in 2-LISP the expression (CONS 1 2) (independent of whether that is ((CONS . 1) . 2) or (CONS . (1 . (2 . NIL))) or whatever) is semantically ill-formed, since no pair can contain a *number* as its CAR or CDR. There is, however, nothing incoherent or problematic about including abstract sequences in our mathematical domain — sequences that might consist of arbitrary entities: s-expressions, mathematical abstractions, or objects from the user's world (a triple, for example, of Thomas Wolsey, the first inaccessible number, and a red-breasted finch). No amount of currying or other structural suggestions deal with the question of how to *designate an abstract sequence.* Nor can we use quoted pairs, such as (+ '(3 4)), or (+ `(, ,Y)), since pairs are reserved for applications, and we are mandated by our aesthetics to avoid category dissonance between structure and semantics.

For all of these reasons, we will define a special structural type, which we will call a ***rail***, to serve as a structural enumerator and normal-form designator of abstract sequences. Rails will in many ways be like the derived lists of 1-LISP, although they are primitive, rather than being implemented in terms of pairs. In particular, as we will illustrate in the next few pages, rails will be defined to embody what we take to be the essence of the original LISP concept of a list.

From an informal point of view, a rail consists of a number of elements, each of which is in turn an s-expression. The elements are numbered starting with the ***index*** 1; each rail has a ***length*** that is the number of elements in it. Thus if a rail R_1 is of length 7, then its seventh element is its last. From a rail each of its elements are accessible, although the reverse accessibility relationship does not hold. Rails are notated (in a manner derived from the old 1-LISP notation for lists) by enclosing the sequence of notations for their elements within square brackets. From the point of view of *declarative* semantics, a rail designates the abstract sequence of objects designated by each of the elements of the rail, respectively. Procedurally, some rails are normal-form sequence designators; thus rails will normalise to rails (this will be explained further in a moment). From just these facts the 2-LISP rail looks similar to MDL's lists and NIL's vector, but we will distinguish them in a moment. Thus we have, as a first approximation to a characterisation (this is rather

imprecise, but we will improve it presently):

$$\textit{SEQUENCES} \equiv \{ Q \mid Q \text{ is a sequence of elements of } D \} \quad \text{(S4-53)}$$

$$\textit{RAILS} \equiv \{ R \mid R \text{ is a rail} \} \quad \text{(S4-54)}$$

$$\text{L-rail} ::= \text{"["_[formula_]}^{*}\ \text{"]"} \quad \text{(S4-55)}$$

$$\Theta_L(L \mid L = \text{"["_formula}_1\text{_formula}_2\text{_ ... _formula}_k\text{_"]"}) \quad \text{(S4-60)}$$
$$= R \in \textit{RAILS} \mid [\forall i\ 1 \le i \le k \text{ the } i\text{'th element of } R \text{ is } \Theta_L(\text{formula}_i)]$$

For example, the rail notated with the string "[1 2 3 4]" designates the abstract sequence of the first four positive integers.

We will assume two functions in our meta-language: one called LENGTH, which is a function from rails and sequences onto their lengths (which may be either finite or infinite), and a selector function called NTH that takes an index and a rail and yields the element at that position. The types of these new functions are as follows:

$$\text{LENGTH} : [[\ \textit{RAILS} \cup \textit{SEQUENCES}\ ] \rightarrow [\ \textit{INTEGERS} \cup \{\infty\}\ ]] \quad \text{(S4-61)}$$
$$\text{NTH} : [[\textit{INTEGERS} \times [\ \textit{RAILS} \cup \textit{SEQUENCES}\ ]] \rightarrow D\ ]$$

We can then begin to characterise the declarative semantics of rails as follows:

$$\forall E \in \textit{ENVS},\ F \in \textit{FIELDS},\ R \in \textit{RAILS} \quad \text{(S4-62)}$$
$$[[\Phi EF(R) = Q] \supset$$
$$[[Q \in \textit{SEQUENCES}\ ] \wedge$$
$$[\text{LENGTH}(Q) = \text{LENGTH}(R)] \wedge$$
$$[\forall i\ 1 \le i \le \text{LENGTH}(Q)\ [Q^{i} = \Phi EF(\text{NTH}(i, R))]]]]$$

This equation is lacking, however, because it does not take up the crucial questions of identity of rails. Since identity hinges on discriminable difference, which in turn hinges on modifiability, we need first to ask in what ways rails can be altered. In the spirit of pairs, we will posit that any element of a rail may be changed (corresponding, in a sense, to the use of 1-LISP RPLACA on lists), and also that the ***tail*** of a rail may be changed, where by the Nth tail of a rail we refer to the (sub)rail beginning *after* the Nth element. Thus if rail R is notated as [2 4 6 3 10], then the second tail of R is [6 8 10], and the fifth tail of R is the empty rail []. Thus we are saying that rails are piece-wise composite; a rail is formed (as a 1-LISP list was) of an element and a tail. It is this structural technique that will allow us to preserve the character of standard LISP lists (and will also distinguish our notion from the more common programming language construct of a vector or one-dimensional array).

In addition, we will say that the *zeroeth* tail of a rail is itself: thus the zeroeth tail of the R of the previous paragraph is [2 4 6 8 10].

(It should be observed that the convention we have adopted specifies that the Nth tail begins with the "Nth plus 1" element, rather than with the Nth; thus the *third* tail of a rail starts with the *fourth* element, not with the third, for example. This may initially seem odd, but it turns out to be the happiest of the available choices. It has the consequence, in particular, that a rail consists of K elements and the Kth tail; for a rail of length N, the consituent tails are the zeroth (the rail itself) through the Nth (the empty rail). And so on and so forth. Further examples will appear in the next pages.)

We will presently define two primitive side-effecting procedures RPLACN and RPLACT (analogous to 1-LISP's RPLACA and RPLACD for lists), which change, respectively, arbitrary elements and arbitrary tails of rails. It is the behaviour of these primitive procedures, and the consequences of the side effects they effect, that most blatantly reveals the identity of 2-LISP rails. The intuition we will attempt to honour throughout is to rationally reconstruct the abilities provided in standard LISPs by the derived notion of a list. These identity considerations will require somewhat complex mathematical modelling; to make them plain, therefore, we will not at first present the equations they must satisfy, but will rather introduce them informally and by example.

First, it is clear that distinct 1-LISP lists can have identical elements; similarly, distinct 2-LISP rails will be allowed to have the same elements. It immediately follows that even if rails are determined to be normal-form designators of sequences (which they will be), they cannot be *canonical* normal-form designators, since distinguishable rails can designate the same abstract mathematical sequence. Secondly, it follows that we cannot in the mathematical characterisation of the field identify rails as sequences of their elements, since that would be too coarse-grained a method of individuation. The logical suggestion would be to posit a special class of rails (***RAILS***), and then to define a function from rails and element positions (*indices*) onto arbitrary s-expressions. However this would be too simple, as we will see in a moment.

In 1-LISP one can use combinations of RPLACA and an arbtirary number of CDRs to change any element in a list; in 2-LISP we will, as mentioned, provide a function called RPLACN, so that the normalisation of expressions of the form

```
(RPLACN <N> <RAIL> <NEW-ELEMENT>)                                    (S4-63)
```

will change the field so that the Nth element of <RAIL> will be <NEW-ELEMENT>. It is also the case in 1-LISP, however, that one can change an arbitrary tail of (most) lists by using RPLACD. Corresponding to this facility in 2-LISP we will say that one can change an arbitrary tail of a rail R by using a primitive procedure called RPLACT. In particular, normalising expressions of the form

```
(RPLACT <N> <RAIL> <NEW-TAIL>)                                       (S4-64)
```

will change the field so that the Nth tail of <RAIL> will henceforth be <NEW-TAIL>. This facility has a number of consequences. First, it means that the length of a given rail may not be constant over the course of a computation; after processing the expression in S4-64, for example, the new length of <RAIL> will be <N> + LENGTH(<NEW-TAIL>), regardless of what it was before. Second, there are considerable consequences to the fact that the <N> in S4-64 can be 0 — which means that two rails that were different (non-EQ, in 1-LISP terminology) can be rendered the same (EQ) in virtue of executing a primitive procedure. This facility, however, cleans up an inelegance in LISP's lists, in which replacing an arbitrary tail starting with any element *other than the first* had different consequences than changing the first. This difference is indicated in the following two "sessions" with 1-LISP and 2-LISP, respectively (user input is, as always, italicised):

```
> (SETQ X '(A B C D))                ; This is 1-LISP                 (S4-66)
> (A B C D)
> (SETQ Z '(L M N O))
> (L M N O)
> (RPLACD (CDR X) Z)                 ; Make 2nd tail of X into Z
> (B L M N O)
> X
> (A B L M N O)
> (PROGN (RPLACA Z 'HELLO)           ; Change the 1st and 2nd elements of Z
         (RPLACA (CDR Z) 'THERE))
> (THERE N O)
> X
> (A B HELLO THERE N O)              ; X sees both changes
> (SETQ Z '(T U V W))
> (T U V W)
> (PROGN (RPLACA X (CAR Z))          ; Make 0'th tail of X into Z
         (RPLACD X (CDR Z))          ; in the only way 1-LISP allows
> (T U V W)
> X
> (T U V W)                          ; X now looks like Z, but its not EQ!
> (PROGN (RPLACA Z 'HELLO)           ; Again, change the 1st and 2nd
         (RPLACA (CDR Z) 'THERE))    ; elements of Z
> (THERE V W)
```

```
> X                                  ; X sees the 2nd change, but not
> (T THERE V W)                      ; the 1st.
```

There is no way, in other words, to change a 1-LISP list to be identical to another, in such a way that any subsequent changes on the latter will be seen in the former. In distinction, the definition we have posited for 2-LISP would engender the following (2-LISP quote marks have approximately the same meaning as in 1-LISP, but ignore for now the fact that the replies made by the system are quoted as well):

```
> (SET X '[A B C D])                 ; This is 2-LISP                (S4-66)
> '[A B C D]                         ; SET is like 1-LISP's SETQ.
> (SET Z '[L M N O])
> '[L M N O]
> (RPLACT 2 X Z)                     ; Make 2nd tail of X into Z
> '[L M N O]
> X
> '[A B L M N O]
> (BLOCK (RPLACN 1 Z 'HELLO)         ; Change the 1st and 2nd elements of Z
         (RPLACN 2 Z 'THERE))
> 'THERE
> X
> '[A B HELLO THERE N O]             ; X sees both changes
> (SET Z '[T U V W])
> '[T U V W]
> (RPLACT 0 X Z)                     ; Make 0'th tail of X into Z
> '[T U V W]
> X
> '[T U V W]                         ; X and Z are now the same rail.
> (BLOCK (RPLACN 1 Z 'HELLO)         ; Again, change the 1st and 2nd
         (RPLACN 2 Z 'THERE))        ; elements of Z
> 'THERE
> X
> '[HELLO THERE V W]                 ; X sees both changes again.
```

2-LISP RPLACT is also defined to be able to *add* elements, in the following formal sense: the index to RPLACT must be between 0 and *the length of the rail.* This again clears up an oddity about 1-LISP's RPLACD, as shown in the following parallel sessions:

```
> (SETQ X '(A B))                    ; This is 1-LISP                (S4-67)
> (A B)                              ; Make X a 2-element list
> (SETQ Y '())                       ; Make Y a 0-element list
> NIL                                ; () is NIL, of course.
> (RPLACD (CDR X) '(C D))            ; Set the 2nd tail to be (C D)
> (B C D)
> X
> (A B C D)                          ; A length 2 list can be extended.
> (RPLACD Y '(C D))                  ; A length 0 list, however, cannot
> <ERROR>                            ; be extended.
> Y
> NIL                                ; Y is still '()
```

In other words 1-LISP lists of length 0 are quite different from lists of other lengths (they are all, in particular, the same atom NIL, whereas there can be arbitrary numbers of distinct lists of any other length). In 2-LISP, however, we have the following symmetry over rails of any length:

```
> (SET X '[A B])                    ; This is 2-LISP                (S4-68)
> '[A B]                            ; Make X a 2-element rail
> (SET Y '[])                       ; Make Y a 0-element rail
> '[]
> (RPLACT 2 X '[C D])               ; Set the 2nd tail to be [C D]
> '[C D]
> X
> '[A B C D]                        ; A length 2 rail can be extended.
> (RPLACT 0 Y '[C D])
> '[C D]
> Y                                 ; A length 0 rail can be extended also
> '[C D]
```

As we will see many times in the examples throughout the remainder of the dissertation, this behaviour simplifies a number of otherwise rather tricky coding situations. Though not of great importance in and of itself, the increased clarity is a noticeable convenience. As a final example, to illustrate this, we show a so-called *destructive splicing* procedure that inserts a new list or rail fragment into a pre-existing list or rail. In particular, we will define a procedure called SPLICE, of three arguments: a rail (list) to work on, an element to trigger on, and a new rail to splice into the old one at the first position where the trigger element occurs, if there is one. In addition, we will require that SPLICE return $T or $F depending on whether the splice was actually performed (i.e., on whether an occurrence of the trigger element was found). Thus if X designates the rail

```
[DO YOU MEAN * WHEN YOU SAY *]                                    (S4-71)
```

then we would expect

```
(SPLICE '[YOU ARE HAPPY] '* X)                                    (S4-72)
```

to return $T with X now designating the rail

```
[DO YOU MEAN YOU ARE HAPPY WHEN YOU SAY *]                        (S4-73)
```

The 2-LISP definition is as follows (this definition modifies the inserted fragment; if this were not intended, the line [[NEW COPY NEW)]] could be inserted as a second binding in the LET on the fifth line; such a COPY is defined in S4-333, below):

```
(DEFINE SPLICE                                                    (S4-74)
   (LAMBDA [NEW TRIGGER OLD]
      (COND [(EMPTY OLD) $F]
            [(= (NTH 1 OLD) TRIGGER)
             (LET [[OLD-TAIL (TAIL 1 OLD)]]
                  (BLOCK (RPLACT 0 OLD NEW)
                         (RPLACT (LENGTH NEW) NEW OLD-TAIL)
                         $T))]
            [$T (SPLICE NEW TRIGGER (TAIL 1 OLD))])))
```

After checking for the appropriate terminating condition, SPLICE checks to see whether the first element is the trigger, and if so splices in a copy of the new rail. The binding of OLD-TAIL is necessary since otherwise, after processing (RPLACT 0 OLD NEW), there would be no way to refer to the tail of the original OLD. If the first element is not the trigger, it iterates down the rail until it either finds a copy of the trigger, or exhausts OLD. The techniques are of course standard.

Because of the (RPLACT 0 ...), we do not need to retain two "trailing pointers", one to use for the modifications when the other indicates an appropriate element has been found. Note as well that no special care needs to be taken for an empty NEW; (SPLICE '[] <A> <B>) would remove the first instance of element <A> from <B>. Thus (SPLICE '[] 'NOT '[I DID NOT NOT ANSWER YOU]) would change the third argument to be '[I DID NOT ANSWER YOU].

For contrast, we can construct an analogous definition of SPLICE in 1-LISP. A natural first attempt would be as follows. Because we need to use RPLACD, which operates on CDRs, we have to break it into two parts; one to test for the *first* element of OLD being the trigger, and another part that allows us to use two trailing pointers:

```
(DEFINE SPLICE1                                                    (S4-76)
   (LAMBDA (NEW TRIGGER OLD)
      (COND ((NULL OLD) NIL)
            ((EQ (CAR OLD) TRIGGER)
             (BLOCK (RPLACD (NTHCDR (- (LENGTH NEW) 1) NEW)
                            (CDR OLD))
                    T))
            ((NULL (CDR OLD)) NIL)
            (T (SPLICE-HELPER1 NEW TRIGGER OLD)))))

(DEFINE SPLICE-HELPER1                                             (S4-76)
   (LAMBDA (NEW TRIGGER OLD)
      (COND ((EQ (CADR OLD) TRIGGER)
             (BLOCK (RPLACD OLD NEW)
                    (RPLACD (NTHCDR (- (LENGTH NEW) 1) NEW)
                            (CDDR OLD))
                    T))
            ((NULL (CDDR OLD)) NIL)
            (T (SPLICE-HELPER1 NEW TRIGGER (CDR OLD))))))
```

However in spite of its increased complexity, there are two ways in which this 1-LISP version of SPLICE is not as general as the 2-LISP version in S4-74. First, $SPLICE_1$ will fail if NEW is empty (i.e. is NIL or ()), since (NTHCDR -1 NIL) will cause an error (we assume NTHCDR returns its whole second argument if its first argument is 0, the CDR of its second argument if its first argument is 1, and so forth). This case could be checked explicitly as follows:

```
(DEFINE SPLICE2                                                    (S4-77)
   (LAMBDA (NEW TRIGGER OLD)
      (COND ((NULL OLD) NIL)
            ((EQ (CAR OLD) TRIGGER)
             (IF (NULL NEW)
                 T
                 (BLOCK (RPLACD (NTHCDR (- (LENGTH NEW) 1) NEW)
```

```
                                (CDR OLD))
                        T)))
            ((NULL (CDR OLD)) NIL)
            (T (SPLICE-HELPER2 NEW TRIGGER OLD)))))

(DEFINE SPLICE-HELPER2                                                    (S4-78)
   (LAMBDA (NEW TRIGGER OLD)
      (COND ((EQ (CADR OLD) TRIGGER)
             (BLOCK (IF (NULL NEW)
                        (RPLACD OLD (CDDR OLD)))
                        (BLOCK (RPLACD OLD NEW)
                               (RPLACD (NTHCDR (- (LENGTH NEW) 1) NEW)
                                       (CDDR OLD))))
                    T))
            ((NULL (CDDR OLD)) NIL)
            (T (SPLICE-HELPER2 NEW TRIGGER (CDR OLD))))))
```

But even still there is a problem: if the *first* element of the original OLD is the trigger, then the explicit check for that in $SPLICE_2$ fails to make the change visible to others who have pointers to OLD. This is particularly obvious where NEW is NIL, where we return T but do nothing (that alone ought to make us suspicious), but it is a problem in any case. In order to compensate for this inability, the practice is typically to have procedures like SPLICE return the modified list, so that a user can reset variables explicitly. Thus a typical call to SPLICE might be:

```
(SETQ SAYING (SPLICE SAYING '* INSERT))                                   (S4-79)
```

We could modify SPLICE to return the appropriate list:

```
(DEFINE SPLICE3                                                           (S4-80)
   (LAMBDA (NEW TRIGGER OLD)
      (COND ((NULL OLD) OLD)
            ((EQ (CAR OLD) TRIGGER)
             (IF (NULL NEW)
                 (CDR OLD)
                 (BLOCK (RPLACD (NTHCDR (- (LENGTH NEW) 1) NEW)
                                (CDR OLD))
                        NEW)))
            ((NULL (CDR OLD)) OLD)
            (T (BLOCK (SPLICE-HELPER3 NEW TRIGGER OLD)
                      OLD)))))

(DEFINE SPLICE-HELPER3                                                    (S4-81)
   (LAMBDA (NEW TRIGGER OLD)
      (COND ((EQ (CADR OLD) TRIGGER)
             (IF (NULL NEW)
                 (RPLACD OLD (CDDR OLD))
                 (BLOCK (RPLACD OLD NEW)
                        (RPLACD (NTHCDR (- (LENGTH NEW) 1) NEW)
                                (CDDR OLD)))))
            ((NULL (CDDR OLD)) NIL)
```

```
(T (SPLICE-HELPER3 NEW TRIGGER (CDR OLD))))))
```

However now we have lost the bit of information that was originally returned saying whether or not the trigger was found. Our final version returns a list of two elements; the first is the flag (T or NIL) saying whether the trigger was found; the second element is the possibly modified list (instead of a two-element list, a "multiple-value return" mechanism might be used here, of the sort explored below in section 5.d.i). Thus one might use $SPLICE_4$ as follows:

```
(LET ((PAIR (SPLICE SAYING '* INSERT)))                          (S4-82)
   (SETQ SAYING (CADR PAIR))
   ... some use of (CAR PAIR) as flag ... )
```

The full definition is as follows. Note that whereas $SPLICE_4$ returns a two-element list as just agreed, $SPLICE\text{-}HELPER_4$ returns only the flag T or NIL depending on whether or not the insertion was effected; if $SPLICE\text{-}HELPER_4$ gets called at all, OLD is always the appropriate list to return.

```
(DEFINE SPLICE4                                                  (S4-83)
   (LAMBDA (NEW TRIGGER OLD)
      (COND ((NULL OLD) (LIST NIL OLD))
            ((EQ (CAR OLD) TRIGGER)
             (LIST T
                   (IF (NULL NEW)
                       (CDR OLD)
                       (BLOCK (RPLACD (NTHCDR (- (LENGTH NEW) 1) NEW)
                                      (CDR OLD))
                              NEW))))
            ((NULL (CDR OLD)) (LIST NIL OLD))
            (T (LIST (BLOCK (SPLICE-HELPER4 NEW TRIGGER OLD)
                            OLD))))))

(DEFINE SPLICE-HELPER4                                           (S4-84)
   (LAMBDA (NEW TRIGGER OLD)
      (COND ((EQ (CADR OLD) TRIGGER)
             (BLOCK (IF (NULL NEW)
                        (RPLACD OLD (CDDR OLD))
                        (BLOCK (RPLACD OLD NEW)
                               (RPLACD (NTHCDR (- (LENGTH NEW) 1) NEW)
                                       (CDDR OLD))))
                    T))
            ((NULL (CDDR OLD)) NIL)
            (T (SPLICE-HELPER4 NEW TRIGGER (CDR OLD))))))
```

Though we won't consider this example further, the lessons are presumably clear. First, the 1-LISP version was an order of magnitude more difficult than the 2-LISP version of S4-74, both in resultant complexity, in difficulty of design, in possibility of error, and so

forth. It is also more difficult to use, because the list has to be passed back. We may observe, furthermore, that the complexity was due to a particular kind of circumstance: boundary conditions and the avoidance of fence-post errors (the *general* case was handled in much the same way in both dialects). In particular, what distinguished the 2-LISP version from the 1-LISP version were the limiting cases — an empty NEW or an instance of TRIGGER in first position. While they were adequately treated by the general code in 2-LISP, they had to be handled specially, with some awkwardness, in 1-LISP.

Another simple example of the same type involves pushing an element onto the front of a stack or queue. Whereas a simple PUSH can be defined in 2-LISP using RPLACT 0, so that expressions of the following sort:

```
(PUSH <NEW-ELEMENT> <STACK>)                                    (S4-85)
```

will change STACK in such a way that anyone who now inquires after its first element will see NEW-ELEMENT, it is difficult to generate this behaviour in 1-LISP. The problem is that if CONS is used, then the modified stack or queue has to be returned and, if necessary, the main pointer to the stack reset appropriately. It might seem possible to use RPLACA, except then the stack cannot be allowed to become empty, since a subsequent RPLACA would fail.

It is worth pausing, for a moment, to consider why we care. Some readers may think it is a waste of time to pursue aesthetic issues in what ought to be an analytic investigation, but that is not our view. We are in this chapter designing a specific formalism; a formalism that we will use heavily in the chapters ahead (chapter 3, in contrast, was analytic and paid no attention to design, except minimally in defining meta-linguistic conventions). We will not explore, in this kind of fine detail, the consequences of the many small design issues that have been faced in specifying 2-LISP (the requirement that each s-expression have a single handle, as is set out in the next section, is another choice much like the present one: seemingly innocent but of tremendous import in reflective work). Nonetheless, we do well to appreciate their potential impact, particularly in consort. These considerations are particularly germane as we reach towards reflection, for reflective code is rather subtle, and we must avail ourselves of all the aesthetic help we can muster along the way (especially the kernel that is our subject matter — user programming that employs this kernel may well be simpler).

But to return to specification. Our mathematical characterisation must support the behaviour manifested in all these examples, since, as is the case in any design, it is the desired behaviour that drives the identity conditions, rather than the other way around. The solution will be to define two mutable first-order asymmetric relationships over rails, the first mapping a rail onto its first element (if it has one), and the second mapping a rail onto its first tail (again if it has one). This approach is very like the standard way in which 1-LISP lists are described in terms of a *first* and *rest*, but of course we have the luxury of defining our meta-theoretic first and rest functions in non-constructive ways.

$$\textit{SEQUENCES} \equiv \{\ Q \mid Q \text{ is a sequence of elements of } D\ \} \qquad \text{(S4-86)}$$

$$\textit{RAILS} \equiv \{\ R \mid R \text{ is a rail}\ \} \qquad \text{(S4-87)}$$

We first define two primitive function classes in the meta-language — classes in order to handle the mutability:

$$\begin{aligned} \textit{FIRSTS} &\equiv [\ \textit{RAILS} \rightarrow [\ S \cup \{\perp\}\ ]] \\ \textit{RESTS} &\equiv [\ \textit{RAILS} \rightarrow [\ \textit{RAILS} \cup \{\perp\}\ ]] \end{aligned} \qquad \text{(S4-88)}$$

These are entirely parallel to the *CARS* and *CDRS* function classes we are adopting from 1-LISP:

$$\begin{aligned} \textit{CARS} &\equiv [\ \textit{PAIRS} \rightarrow S\ ] \\ \textit{CDRS} &\equiv [\ \textit{PAIRS} \rightarrow S\ ] \end{aligned} \qquad \text{(S4-89)}$$

In addition, this is an appropriate time to add the property-list concept to 2-LISP; we will assume that all property lists are rails; thus we have:

$$\textit{PROPS} \equiv [\ \textit{ATOMS} \rightarrow \textit{RAILS}\ ] \qquad \text{(S4-90)}$$

Thus we have the following tentative definition of the 2-LISP set of possible fields (this is too broad, as we will show in a moment):

$$\textit{FIELDS} \equiv [\ \textit{CARS} \times \textit{CDRS} \times \textit{FIRSTS} \times \textit{RESTS} \times \textit{PROPS}\ ] \qquad \text{(S4-91)}$$

Because there are five of these, which are harder to remember than the simple 3 we used in 1-LISP, we will define five meta-theoretic utility functions:

$$\begin{aligned} \text{CAR} &\equiv \lambda S\ .\ \lambda F\ .\ [F^1(S)] \\ \text{CDR} &\equiv \lambda S\ .\ \lambda F\ .\ [F^2(S)] \\ \text{FIRST} &\equiv \lambda S\ .\ \lambda F\ .\ [F^3(S)] \\ \text{REST} &\equiv \lambda S\ .\ \lambda F\ .\ [F^4(S)] \\ \text{PROP} &\equiv \lambda S\ .\ \lambda F\ .\ [F^6(S)] \end{aligned} \qquad \text{(S4-92)}$$

Thus these functions in the meta-language take two arguments, whereas the corresponding embedded procedures within the dialect do not need the field as an explicit argument.

The first thing that we must do is to revise our definition of 2-LISP fields, since rails either do or do not have firsts and rests together. In particular, we have the following constraint:

$$\forall R \in \textit{RAILS},\ F \in \textit{FIELDS} \qquad \text{(S4-93)}$$
$$[[[\ \exists S_f \in S\ [\mathrm{FIRST}(R,F) = S_f]\ ] \wedge [\ \exists R_r \in \textit{RAILS}\ [\mathrm{REST}(R,F) = R_r]\ ]] \vee$$
$$[[\ \mathrm{FIRST}(R,F) = \bot\ ] \wedge [\ \mathrm{REST}(R,F) = \bot\ ]]]$$

Therefore we will define *FIELDS* as follows:

$$\textit{FIELDS} \equiv \{\ F \in [\ \textit{CARS} \times \textit{CDRS} \times \textit{FIRSTS} \times \textit{RESTS} \times \textit{PROPS}\ ] \qquad \text{(S4-94)}$$
$$[[\ \forall R \in \textit{RAILS}$$
$$[[[\ \exists S_f \in S\ [\mathrm{FIRST}(R,F) = S_f]\ ] \wedge$$
$$[\ \exists R_r \in \textit{RAILS}\ [\mathrm{REST}(R,F) = R_r]\ ]] \vee$$
$$[[\ \mathrm{FIRST}(R,F) = \bot\ ] \wedge [\ \mathrm{REST}(R,F) = \bot\ ]]]]\}$$

Given all of this machinery, we can define a new length and a selector function (these supercede the initial versions we defined in S4-91) that will ultimately enable us to define the appropriate behaviours for RPLACN and RPLACT in section 5.b. Note that the definition supports infinite-length rails (such as circular ones, for example), and the fact that NTH is defined over such rails as well as over finite ones.

$$\mathrm{NTH}\ :\ [[\ \textit{INTEGERS} \times \textit{RAILS} \times \textit{FIELDS}\ ] \rightarrow [\ S \cup \{\bot\}\ ]] \qquad \text{(S4-95)}$$
$$\equiv \lambda I, R, F\ .\ [\ \textit{if}\ [\mathrm{FIRST}(R,F) = \bot]$$
$$\textit{then}\ \bot$$
$$\textit{elseif}\ [I = 1]\ \textit{then}\ \mathrm{FIRST}(R,F)$$
$$\textit{else}\ \mathrm{NTH}(I-1, \mathrm{REST}(R,F), F)\ ]$$

$$\mathrm{LENGTH}\ :\ [[\ \textit{RAILS} \times \textit{FIELDS}\ ] \rightarrow [\ \textit{INTEGERS} \cup \{\infty\}\ ]] \qquad \text{(S4-96)}$$
$$\equiv \lambda R, F\ .\ [\ \textit{if}\ [\mathrm{FIRST}(R,F) = \bot]$$
$$\textit{then}\ 0$$
$$\textit{elseif}\ [\ \exists N\ [\mathrm{NTH}(N,R,F) = \bot\ ]]$$
$$\textit{then}\ [1 + \mathrm{LENGTH}(\mathrm{REST}(R,F),F)]$$
$$\textit{else}\ \infty$$

We can use these functions to define the notational interpretation function for rails:

$$\text{L-rail} \ ::= \ \text{"["_[formula_]}^{*}\ \text{"]"} \qquad \text{(S4-97)}$$

$$\Theta_L(L \mid L = \text{"["}_\,\mathrm{formula}_1\,_\,\mathrm{formula}_2\,_\,\ldots\,_\,\mathrm{formula}_k\,_\,\text{"]"}) \qquad \text{(S4-98)}$$
$$= R \in \textit{RAILS} \mid [[\forall i\ 1 \le i \le k\ \mathrm{NTH}(i,R,F) = \Theta_L(\mathrm{formula}_i)\ ] \wedge$$
$$[\ \mathrm{NTH}(k+1,R,F) = \bot\ ]$$

In addition, we can review our claim first that rails designate sequences; then, that rails are normal-form designators; and third that they satisfy the requirements on such normal-form designators. These last two observations are more complex in the case of rails than with previous examples, because a rail is in normal-form only if its elements are in normal form. In the next few sets of equations we will express these constraints. Note that we assume a single-argument LENGTH defined over abstract sequences, as well as our recently introduced two-argument one defined over structural rails. First we specify that rails designate the sequence of objects designated by their elements:

```
∀E ∈ ENVS, R ∈ RAILS, F ∈ FIELDS                                    (S4-99)
     [[ ΦEF(R) = Q ] ⊃
      [[ Q ∈ SEQUENCES ] ∧
       [ LENGTH(Q) = LENGTH(R,F) ] ∧
       [ ∀i 1≤i≤LENGTH(Q) [ Q^i = ΦEF(NTH(i,R,F)) ]]]]
```

In order to define the procedural consequence of rails, it is helpful to define an auxiliary function NFD (for "normal-form-designator"):

```
NFD  : [[ S × D ] → {Truth, Falsity} ]                              (S4-100)
     ≡ λS, D . [[ ∀E ∈ ENVS, F ∈ FIELDS [ ΦEF(S) = D ]] ∧
                 [ NORMAL-FORM(S) ]]
```

Thus for example [NFD("3,3)] is true, whereas [NFD("(+ 1 2),3)] and [NFD("$F,3)] are false.

As is by now our custom, we will first set down the *desideratum* we would ultimately like to prove regarding rails: that all expressions that designate sequences will (and that normalise at all) normalise to a rail:

```
∀S1, S2 ∈ S, E ∈ ENVS, F ∈ FIELDS                                   (S4-101)
     [[[ ΦEF(S1) ∈ SEQUENCES ] ∧ [ S2 = ΨEF(S1) ]] ⊃
      [[ S2 ∈ RAILS ] ∧
       [ LENGTH(S2,F) = LENGTH(ΦEF(S1)) ] ∧
       [ ∀i, 1≤i≤LENGTH(S2,F) [ NFD(NTH(i,S2,F), [ΦEF(S1)]^i ) ]]]]
```

We can now specify in particular the procedural consequence of rails. First, we show that rails that are in normal form are self-normalising:

```
∀E ∈ ENVS, R ∈ RAILS, F ∈ FIELDS, C ∈ CONTS                         (S4-102)
     if [ ∀i 1≤i≤LENGTH(R) [ NORMAL-FORM(NTH(i, R)) ]]
       then [ Σ(R, F, E, C) = C(R,ΦEF(R),E,F) ]]
```

This last equation is stronger even that the behaviour implied (but not yet proved) by S4-101, because S4-102 claims that if a rail is in normal-form, it will normalise to itself, whereas the prior equations merely assert that if a rail is in normal-form it will normalise to some rail also in normal form, possibly different.

The general computational significance of rails is more difficult to specify than any of the special cases treated so far, because of potential side-effects engendered by the normalising of the interior elements (we must also make explicit the fact that the normalisation of infinite-length rails will not terminate). It can, however, be recursively defined, since the first tail of any non-empty rail is itself a rail. As a first attempt we have:

```
∀R ∈ RAILS, E ∈ ENVS, F ∈ FIELDS, C ∈ CONTS                    (S4-103)
    [Σ(R,E,F,C) = if [LENGTH(R,F) = ∞] then ⊥
                  elseif [NTH(1,R,F) = ⊥] then C(R,<>,E,F)
                  else Σ(NTH(1,R,F),E,F,
                         [λ<S1,D1,E1,F1> .
                            Σ(REST(R,F),E1,F1,
                              [λ<R2,D2,E2,F2> . C(S,D,E2,F2)])])]
```

There are however several problems. First, we have not specified the S and D in the last call to C; the idea is that S is the rail whose first element is S_1 and whose first tail is R_2. Similarly, D is intended to be the sequence whose first element is D_1 and whose remainder is D_2. Finally, rather than F_2 being passed back as the final field, a field should be returned that encodes these new first and rest relationships. It is easier to state these relationships as constraints than to modify the main definition:

```
∀R ∈ RAILS, E ∈ ENVS, F ∈ FIELDS, C ∈ CONTS                    (S4-104)
   [Σ(R,E,F,C) = if [LENGTH(R,F) = ∞] then ⊥
                 elseif [NTH(1,R,F) = ⊥] then C(R,<>,E,F)
                 else Σ(NTH(1,R,F),E,F,
                        [λ<S1,D1,E1,F1> .
                           Σ(REST(R,F),E1,F1,
                             [λ<R2,D2,E2,F2> . C(S,D,E2,F3)])])]
      where S ∈ RAILS and D ∈ SEQUENCES;
            NTH(1,S,F3) = S1;
            REST(S,F3) = R2;
            F3 = F2 otherwise;
            D^1 = D1;
            ∀i 1≤i≤LENGTH(D2) [ D^(i+1) = D2^i]
```

We are all but done; there is, however, one remaining problem. S4-104 as presented does not ensure that normal-form rails, other than empty ones, are self-normalising. Thus we need one additional clause at the outset stating that. In addition, we need to modify the account so that empty rails are not returned if they pass the normal-form filter, since

otherwise (this is explored further in section 4.b.v) modifying the last tail to a normalised rail would modify the original rail as well.

```
∀R ∈ RAILS, E ∈ ENVS, F ∈ FIELDS, C ∈ CONTS                          (S4-105)
  [Σ(R,E,F,C) =
      if [∀i 1≤i≤LENGTH(R,F) [NORMAL-FORM(NTH(i,R,F))]
         then C(R,D0,E,F)
         elseif [LENGTH(R,F) = ∞] then ⊥
         elseif [NTH(1,R,F) = ⊥]
                then C("[],<>,E,F) where "[] is inaccessible in F
                else Σ(NTH(1,R,F),E,F,
                       [λ<S1,D1,E1,F1> .
                          Σ(REST(R,F),E1,F1,
                            [λ<R2,D2,E2,F2> . C(S,D,E2,F3)])])
       where S ∈ RAILS and D ∈ SEQUENCES and D0 ∈ SEQUENCES;
             NTH(1,S,F3) = S1;
             REST(S,F3) = R2;
             F3 = F2 otherwise;
             LENGTH(R,F) = LENGTH(D0) = LENGTH(D);
             D^1 = D1;
             ∀i 1≤i≤LENGTH(D2) [ D^(i+1) = D2^i ];
             ∀i 1≤i≤LENGTH(D0) [ D0^i = ΦEF(NTH(i,R,F)) ]
```

It should be noted that all of these issues of identity are defined with respect to rails; *sequence* identity we derive from mathematics: two sequences are the same just in case they contain the same elements in the same order.

It would be possible to define a notion of *type-equivalence* over rails, in the spirit of the type-equivalence we defined in 1-LISP over lists. But we will not do this, because of a striking fact that emerges from our semantical bent; *equality over designated sequences is a coarser-grained identity than that over sequence designators.* One can even speculate that in 1-LISP's type-identity predicate EQUAL there lies an attempt to establish identity of the orderings of objects that the 1-LISP lists encode. The 2-LISP identity predicate is spelled "=" (it will be defined in section 4.b.iii below); over *structures* it is true just in case the structures are one and the same, whereas over *sequences* it is true just in case they are the same *mathematical* sequence — which is to say, just in case the elements are (recursively) the same and in the same order. Since we *use* rails to designate sequences, we use *handles* (notated with a quote mark) to *mention rails*, and since = is extensional (all of these points will be more fully illustrated below), the following normalisations would hold in 2-LISP:

```
(= [1 2 3] [1 2 3])      ⇒   $T                                      (S4-106)
(= '[1 2 3] '[1 2 3])    ⇒   $F
```

In the remainder of this dissertation this distinction between rail identity and sequence identity — a distinction between *identity of designator* and *identity of the object designated* — will largely serve our purposes; in general no need for a notion of *type-identity over designators* will arise (although in the as-yet unsolved area of the identity of function intension we will look briefly at type-equivalence of rails).

Once again substantive examples of rails will await our definition of procedures defined over them; for the present we are constrained to such simple illustrations as these:

```
[1 2 3]          ⇒   [1 2 3]          (S4-107)
[]               ⇒   []
[[$T][$F]]       ⇒   [[$T][$F]]
```

The final introduction to make regarding rails and sequences has to do with a notational abbreviation. We said above that we were not defining the standard list notation to abbreviate chains of pairs, as in 1-LISP. Instead, we will take lexical expressions of the form

$$\text{"("}_\,\text{formula}_1\,_\,\text{formula}_2\,_\,\ldots\,_\,\text{formula}_k\,\text{")"} \qquad \text{(S4-108)}$$

where $1 \leq K$, as abbreviations for the following:

$$\text{"("}_\,\text{formula}_1\,_\,\text{"."}\,_\,\text{"["}\,_\,\text{formula}_2\,_\,\ldots\,_\,\text{formula}_k\,\text{"]"}\,_\,\text{")"} \qquad \text{(S4-109)}$$

In other words, a sequence of notational expressions within parentheses notates a pair, whose CAR is notated by the first, and whose CDR is a rail of elements notated by the remainder. For example:

```
(+ 2 3)              abbreviates   (+ . [2 3])              (S4-110)
(READ)                    "        (READ . [])
(CAR (CONS 'A 'B))        "        (CAR . [(CONS . ['A 'B])])
```

It follows that the expression "()" is *notationally* ill-formed — in 2-LISP, in other words, there is no NIL, and "()" is its name.

From this convention, and from the equation given in S4-38, it follows that from one point of view, all 2-LISP procedures are called with a single argument, which, if this abbreviation is used, will be a rail of zero or more expressions. It is therefore a convention that all 2-LISP procedures will be defined over *sequences*; we will mean, by the phrase "the number of arguments" taken by a function, the number of elements in the sequence. It is possible to define procedures that do not honour this convention, but all primitive 2-LISP

functions obey the protocol, as will all of the functions we define in our examples.

Note as well that it is impossible to construct an application to a function with no arguments at all, since it is impossible to have a pair that has no CDR.

It should be kept in mind that the foregoing comment is *semantical*: it says that the functions *designated by the CAR of a pair* are by and large defined over *sequences in the semantical domain.* It does not follow from anything that has been said that, from a structural point of view, all functions must be called with a *rail* as the argument sequence designator — that all semantically valid pairs must have rails as their CDRs. It is in fact this very separation between sequences and rails that enables 2-LISP to naturally solve the problem of calling functions with a single expression that designates the entire sequence of arguments. Some simple examples of this flexibility are given in the following (LET is approximately as in 1-LISP, except that rails rather than lists are of course used to encode enumerations — it will be defined below):

```
(LET [[X [4 5]]] (+ . X))          ⇒  9                    (S4-111)

(+ . (TAIL 2 [10 20 30 40]))       ⇒  70                   (S4-112)
```

More such examples will arise in due course.

It is once again appropriate to pause for a methodological comment. The work we did in S4-71 through S4-85, as mentioned earlier, enabled us to obtain a cleaner dialect; the present concern with rails as multiple-argument designators is also aesthetic, but it impinges more directly on our goal of reflection. As we commented in chapter 2, the fact that 1-LISP does not allow arguments to be conveniently objectified required the explicit use of APPLY — a situation we are at pains to avoid, particularly because we are adopting a statically scoped dialect. For example, the expressions given in S4-111 and S4-112, in the corresponding 1-LISP treatment, would have to be written roughly as follows:

```
(LET ((X (LIST 4 5))) (APPLY '+ X))     →  9    ; This is 1-LISP    (S4-113)

(APPLY '+ (NTHCDR 2 '(10 20 30 40)))    →  70                        (S4-114)
```

However these work only because numerals self-evaluate; if the objectified expressions contained variables the dialect would have to be dynamically scoped in order for the meta-structural treatment to work. All in all, we are better off able to avoid these rather complex manoeuvrings.

4.a.vi. Handles

We have so far introduced five structural categories: *numerals*, *booleans*, *rails*, *atoms*, and *pairs*. The first three of these designate abstract mathematical objects (numbers, truth-values, and sequences, respectively); the last two can designate entities of any type, since they are general purpose designators, taking their designation from the context (in the case of atoms) or from the value of a function application (in the case of pairs). The sixth and final 2-LISP structural category is called a *handle*, and designates elements of the structural field. Handles are not unlike quoted expressions in 1-LISP, although they have their own notation and identity conditions.

A handle is an atomic element of the field, with a variety of special properties. First, for every element of ***S*** there is exactly one handle that is the canonical normal-form designator of that element (implying an infinite number of handles, not only because there are an infinite number of elements of ***S*** of other types, but also because this claim recurses, implying that every handle has a handle, and so forth). There is a total function on ***S***, in other words, which in our meta-language we will call the HANDLE function, that takes each element of ***S*** onto its handle. Furthermore, from every element of ***S*** its handle is *locally accessible*; in addition, from every handle its *referent* is also locally accessible. In other words the HANDLE relationship, like the CAR and CDR relationships, is asymmetric, but in two other respects it is unlike the CAR and CDR relationships. First, it is bi-directionally local, whereas CAR and CDR are uni-directionally local. In addition, the CAR and CDR relationships are mutable, whereas the handle relationship is not. Thus HANDLE need not be encoded in the ***FIELDS*** part of our meta-theoretic characterisation.

Each handle is notated with a single quote mark ("'") followed by the notation of its referent. These various properties are summarised in the following equations:

HANDLES ≡ { H | H is a handle } (S4-120)

HANDLE : [***S*** → ***HANDLES***] (S4-121)

HANDLE is the function from elements of the structural field onto their handles. The existence and identity conditions on handles are expressed in the following two equations; it follows that HANDLE^{-1} is a total function on ***HANDLES***.

$$\forall S \in \mathcal{S}\ [\exists H \in \mathit{HANDLES} \quad [H = \mathrm{HANDLE}(S) \wedge \forall J\ [J = \mathrm{HANDLE}(S) \supset J = H]]] \qquad \text{(S4-122)}$$

$$\forall H \in \mathit{HANDLES}\ [\exists S \in \mathcal{S}\ [H = \mathrm{HANDLE}(S)]]$$

As remarked, handles are notated using a lexical form similar to the 1-LISP abbreviation for quoted forms (although in 2-LISP this is a primitive, not an abbreviatory, form):

L-handle ::= "'"_<notation for referent> (S4-123)

$$\forall L \in \mathit{L\text{-}HANDLES}\ [[L = \text{"'"}_L_r] \supset [\Theta_L(L) = \mathrm{HANDLE}(\Theta_L(L_r))]] \qquad \text{(S4-124)}$$

That handles designate their referents in a context-independent way is implied by:

$$\forall E \in \mathit{ENVS},\ F \in \mathit{FIELDS},\ H \in \mathit{HANDLES}\ [\Phi EF(H) = \mathrm{HANDLE}^{-1}(H)] \qquad \text{(S4-126)}$$

Similarly, that handles are designed to be the normal-form designators of s-expressions is captured in:

$$\forall E \in \mathit{ENVS},\ F \in \mathit{FIELDS},\ S \in \mathcal{S} \qquad \text{(S4-126)}$$
$$[[\Phi E^{-1}(S) \in \mathcal{S}] \supset [\Psi EF(S) = \mathrm{HANDLE}(\Phi EF(S))]]$$

Equations S4-120 through S4-125 are independent and posited; S4-126 is a claim we will have to prove in section 4.h. The following, which expresses the fact that handles normalise to themselves, is posited as a first step towards its ultimate proof:

$$\forall E \in \mathit{ENVS},\ F \in \mathit{FIELDS},\ H \in \mathit{HANDLES}\ [\Psi EF(H) = H] \qquad \text{(S4-127)}$$

The normalisation of handles will of course be side-effect free, as well as environment-independent:

$$\forall E \in \mathit{ENVS},\ H \in \mathit{HANDLES},\ F \in \mathit{FIELDS},\ C \in \mathit{CONTS} \qquad \text{(S4-128)}$$
$$[\Sigma(H, \langle F, E, C\rangle) = C(H, \mathrm{HANDLE}^{-1}(H), E, F)]$$

And once again, from these conditions the following summary can be shown to follow:

$$\forall H \in \mathit{HANDLES}\ [\mathrm{NORMAL\text{-}FORM}(H)] \qquad \text{(S4-129)}$$

We just said that all handles normalise to themselves: the 2-LISP processor, in other words, does not "strip the quotes off" of meta-level designators (we shall have to introduce a special mechanism to do that presently). We have in consequence the following:

```
'A              ⇒   'A                                        (S4-130)
'[1 2 3]        ⇒   '[1 2 3]          ; designates a rail of numerals
['1 '2 '3]      ⇒   ['1 '2 '3]        ; designates a sequence of numerals
''''''$T        ⇒   ''''''$T
```

The differences between 2-LISP's handles and the corresponding meta-structural designation facility in 1-LISP are several. First, 1-LISP contained a primitive function called QUOTE — an IMPR described in chaper 2 — in terms of which applications were constructed that, according to the declarative semantics we adopted in chapter 4, designated the referent. 1-LISP notational forms using the single quote mark were notational abbreviations for applications in terms of this function. In contrast, 2-LISP is defined with no such quote *function*, because the relationship between entities and their designators is more inextricably woven into the fundamental distinctions made by the category structure of the dialect itself. It is not difficult to *define* a QUOTE function in 2-LISP, but, as we noticed, it is straightforward to define a quote function in 1-LISP as well, since FEXPRS are a more general meta-structural capability. The 1-LISP definition is as follows:

```
(DEFINE QUOTE1                    ; This is 1-LISP              (S4-131)
   (LAMBDA IMPR (ARG) ARG)
```

Note that the body is simply ARG, not (LIST 'QUOTE ARG), because of 1-LISP's de-referencing evaluator. The corresponding 2-LISP definition of QUOTE is virtually identical:

```
(DEFINE QUOTE2                    ; This is 2-LISP              (S4-132)
   (LAMBDA IMPR [ARG] ARG))
```

However the superficial similarity between these two definitions is misleading: they work for quite different reasons. In S4-131 ARG is bound to the unevaluated argument; evaluation of the body of the definition will look up the binding of ARG, returning as a result that un-evaluated argument — the expression, in other words, that the application in terms of $QUOTE_1$ is taken to *designate*. Thus in the evaluation of ($QUOTE_1$ (+ 2 3)) the variable ARG would be bound to the list (+ 2 3), which be returned as the value. Thus we have:

```
(QUOTE1 (F X))    →    (F X)          ; This is 1-LISP          (S4-133)
```

In S4-132, however (as will become plain later in the chapter), ARG is bound to the handle designating the un-normalised argument; evaluating the body in this case will yield that handle. For example, if ($QUOTE_2$ (+ 2 3)) were normalised, ARG would be bound to the handle '(+ 2 3), which would be returned as the normal-form co-designator of the original application. Thus we have:

```
(QUOTE₂ (F X))    ⇒   '(F X)                ; This is 2-LISP        (S4-134)
```

More illustrative of the difference between the dialects are two *extensional* functions that take an s-expression as an argument and "return" the s-expression that designates the *result of processing it* (i.e., the *value* in the 1-LISP case; the *normal-form* in 2-LISP). In standard LISP dialects such a function has been variously called KWOTE, QUOTIFY, etc., and has the following definition:

```
(DEFINE KWOTE                             ; This is 1-LISP        (S4-135)
   (LAMBDA EXPR (ARG) (LIST 'QUOTE ARG)))
```

The 2-LISP version of KWOTE would be exactly the HANDLE function we have used in the equations above; strikingly, however, it turns out that such a function cannot be defined in 2-LISP. In detail the reasons are messy to set forth, but the reason is quite straightforward: such a procedure involves semantic level crossing in a way that the primitive 2-LISP processor by and large avoids. So flat is normal 2-LISP processing that there is no way to obtain a designator at a different meta-level from that of one's arguments. No way, that is, without primitive help: such a capability, therefore, is provided primitively in a function called NAME (rather than HANDLE because, as we will see in section 4.e, it is more general than HANDLE, though that needn't concern us here).

The most salient difference between 1-LISP and 2-LISP quotation, to return to our original concern, has to do with identity and type. Some examples that use 1-LISP's KWOTE and 2-LISP's NAME functions will illustrate. 1-LISP quoted forms are *pairs*, subject to modification like any other. There can be in addition an arbitrary number of such pairs quoting (designating) the same referent. Though that referent is locally accessible from the pair (either by evaluation or by structural decomposition, we may note), none of those pairs are accessible from the referent. Finally, as remarked in section 3.f.ii, neither "EQ" nor "EQUAL" identity of *designator* reveals the identity of the referent, as is illustrated in the following examples (these are all 1-LISP). First we look at four cases using EQ:

```
1:  (EQ ''3 ''3)                                    →  NIL        (S4-136)
2:  (EQ ''(A B) ''(A B))                            →  NIL
3:  (LET ((X '(A B))) (EQ (KWOTE X) (KWOTE X)))     →  NIL
4:  (LET ((X '(A B)) (Y '(A B)))
       (EQ (KWOTE X) (KWOTE Y)))                    →  NIL
```

In each case EQ returns NIL, but all that this shows is that quoted forms are more *finely* distinguished than their referents: in 1 and 3 the referents are indeed the same, whereas in

2 and 4 the referents are different. EQUAL, however, as the next set of expressions indicates, returns T in each case, since in all instances the referents are *type-identical*, which is the property EQUAL is defined over. The fact that the arguments to EQUAL are structure designators is immaterial: it happens that all quotations of a given structure are type-identical, so that while it works out that EQUAL returns T just in case the referents are type-identical, that is in a sense accidental:

```
1:  (EQUAL ''3 ''3)                                          →  T        (S4-137)
2:  (EQUAL ''(A B) ''(A B))                                  →  T
3:  (LET ((X '(A B))) (EQUAL (KWOTE X) (KWOTE X)))           →  T
4:  (LET ((X '(A B)) (Y '(A B)))
       (EQUAL (KWOTE X) (KWOTE Y)))                          →  T
```

In 2-LISP, on the other hand, there is a single handle per referent, which is locally accessible, is *not* modifiable, is not a pair, and *can* be used to determine the identity of referent (2-LISP's "=" is, like 1-LISP's EQ, an *individual* identity function; there is no need for a 2-LISP *type-identity* function, as the examples demonstrate):

```
1:  (= ''3 ''3)                                              ⇒  $T       (S4-138)
2:  (= ''(A B) ''(A B))                                      ⇒  $F
3:  (LET [[X '(A B)]] (= (NAME X) (NAME X)))                 ⇒  $T
4:  (LET [[X '(A B)] [Y '(A B)]]
       (= (NAME X) (NAME Y)))                                ⇒  $F
```

The 2-LISP NAME function is so often useful in meta-structural work that it has its own lexical abbreviation — one that has appeared from time to time in previous examples: applications in terms of it can be abbreviated using an up-arrow ("↑"). We pronounce this "up"; thus the expression (= ↑X ↑Y) would be read "equal up-X up-Y". Thus example S4-138 can be re-written as follows:

```
1:  (= ''3 ''3)                                              ⇒  $T       (S4-139)
2:  (= ''(A B) ''(A B))                                      ⇒  $F
3:  (LET [[X '(A B)]] (= ↑X ↑X))                             ⇒  $T
4:  (LET [[X '(A B)] [Y '(A B)]] (= ↑X ↑Y))                  ⇒  $F
```

The NAME function, and naming issues in general, will be further explored in section 4.e on meta-structural facilities.

4.a.vii. Category Summary

The foregoing six sections completely define the 2-LISP (and 3-LISP) field. We can summarise the six kinds of structural field element by defining ***S***:

$$S \equiv [\ NUMERALS \cup BOOLEANS \cup ATOMS \cup PAIRS \cup RAILS \cup HANDLES\ ] \qquad \text{(S4-140)}$$

No individual tokens of any of these categories were mentioned, with the exception of $T and $F, the two boolean constants, which deserved mention only because that category is finite. From each element of ***S*** the handle designating that element is accessible; otherwise, from the numerals, booleans, and atoms no other structures were accessible, but from a handle its referent can be reached, from a pair its CAR and its CDR, and from a rail all of its elements and all of its tails.

For completeness, we also reproduce here the definition of the 2-LISP fields:

$$\begin{aligned} FIELDS \equiv \{\ & F \in [\ CARS \times CDRS \times FIRSTS \times RESTS \times PROPS\ ] \\ & [\![\ \forall R \in RAILS \\ & \quad [\![\![\ \exists S_r \in S\ [\ FIRST(R,F) = S_r\ ]\!]\ \wedge \\ & \quad\ [\ \exists R_r \in RAILS\ [\ REST(R,F) = R_r\ ]\!]\!]\ \vee \\ & \quad [\![\ [\ FIRST(R,F) = \bot\ ]\ \wedge\ [\ REST(R,F) = \bot\ ]\!]\!]\!]\} \end{aligned} \qquad \text{(S4-141)}$$

As well as the set ***S***, we define three additional sets of entities which together comprise the remainder of the semantical domain: the first consisting of non-functional mathematical abstractions (numbers, sequences, and truth-values), the second consisting of all of the continuous functions defined over the semantical domain (we will break this up into sub-categories in due course), and the third consisting as usual of the user's world of objects in relationship:

$$\begin{aligned} FUNCTIONS &\equiv \text{the set of continuous functions over } D \\ ABSTRACTIONS &\equiv [\ NUMBERS \cup TRUTH\text{-}VALUES \cup SEQUENCES\ ] \\ USER\text{-}WORLD &\equiv \text{the user's domain of objects in relationship} \end{aligned} \qquad \text{(S4-142)}$$

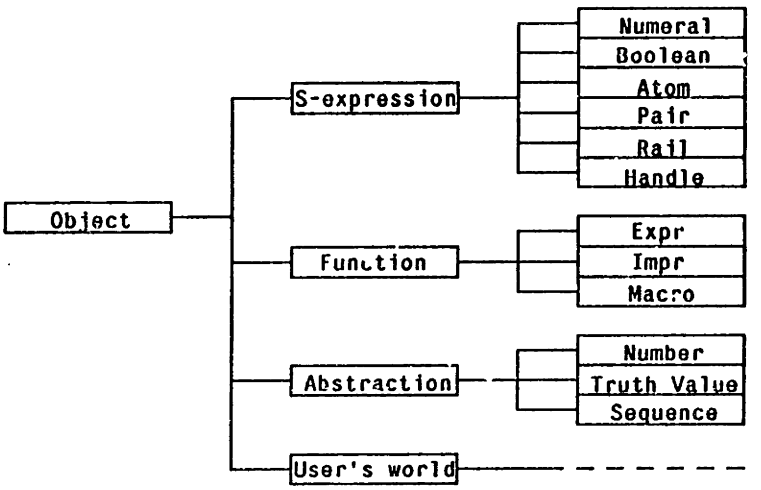
The full semantical domain is the union of these four:

$$D \equiv S \cup FUNCTIONS \cup ABSTRACTIONS \cup USER\text{-}WORLD \qquad \text{(S4-143)}$$

Note that this is a recursive definition of ***D***, since the continuous functions over ***D*** are included in the specification of ***D*** (i.e. $D \subset [\ D \times D\ ]$).

This 2-LISP semantic domain is thus typed as follows:

(S4-144)



4.a.viii. Normal-form Designators

We promised to define 2-LISP's notion of *normal-form* from the category structure of the semantical domain, in contrast with the parallel notion in the λ-calculus, where it is defined in terms of the deductive machinery. Various comments have been made in passing, in this chapter, about normal form designators, but we can summarise them here.

First we take the s-expressions: as we said in section 4.a.vi, handles are normal-form designators of all s-expressions (including, recursively, the handles themselves); handles, further-more, are *canonical* normal-form designators. The following is provable from S4-120 — S4-125:

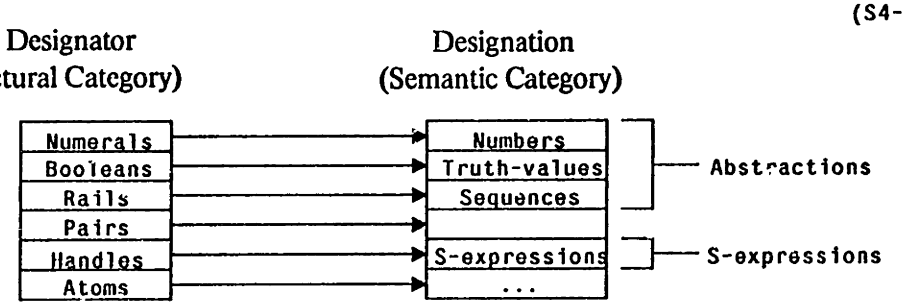
$$\forall S \in \mathcal{S}\ [\mathrm{NFD}(\mathrm{HANDLE}(S),S)] \qquad \text{(S4-146)}$$

With respect to the semantical category structure given in S4-144, a sub-class of the s-expressions has been used as the normal-form designator of all s-expressions. This has the requisite sparseness: the other five structural categories remain available as normal-form designators of the other semantical types (since *all* normal-form designators, tautologically, must be elements of $\mathcal{S}$, whereas we will require normal-form designators for all of $\mathcal{D}$).

Each of the sub-categories of the ***ABSTRACTIONS*** has its own structural category as normal-form designator: the numerals, booleans, and rails, respectively, are normal-form designators of the numbers, truth-values, and sequences. The first two categories are canonical; rails, as discussed above, are not.

Thus so far we have the following normal-form correspondences, where on the left-hand side of the diagram are given the six available structural categories, and on the right is the category structure of the full semantical domain.

(S4-146)



What remains, then, is to identify normal-form designators for the functions and for the user's world.

Two comments are in order. First, all elements of the three of the four categories we have just identified are *always* in normal form: every single handle, numeral, and boolean, in particular, is a normal-form designator, and will therefore normalise to itself. Not all rails, however, are in normal-form: a rail is normal just in case its elements are normal.

Regarding the atoms and the pairs, however, we have indicated above that by and large they are not normal-form designators. We still have the freedom, therefore, to make some of them normal-form designators if we choose, without violating our mandate of category alignment. Another possible aesthetic, of course, would be to require a special collection of structural categories, each of which was the normal-form designator of a particular category of semantical object, but although this is cleaner in one sense than the proposals we will ultimately adopt, it obtains that cleanliness at the expense of rather too many structural categories (simplicity is part of cleanliness).

We said in the previous chapter that some kind of s-expression in the spirit of a closure would serve as a normal-form function designator. The discussion just laid out suggests that either atoms or pairs might serve as the syntactic category for closure. Atoms are too atomic: closures must be structural entities containing information, and atoms, in an informal sense, have no place to store that information. (This last, of course, is not a theoretically justifiable claim, but rather a pragmatic one: from a mathematical point of

view we could simply posit that the atoms, in alphabetic order, would be the normal form designators of the functions computed by some abstract machine — say a Turing machine of a certain form. The semantics of function designators would in that case not be compositional in any way. However we will not pursue such suggestions.) Pairs, however, are not ruled out by this criterion. It is natural, therefore, to review any possible arguments *against* using pairs as normal-form function designators, since they would seem to satisfy all of the design considerations we have explicitly adopted.

The standard concern with making closures out of pairs (or any similar "accessible" type of structure) is that it is inelegant to allow a user to use such functions as CAR and CDR over "functions". Closures, it is often said, can *only be applied*: they should not look like structures open to dissection. It is striking to realise that this concern arises out of the semantical informality of standard LISPs, however, and should not trouble us. In particular, *even though we will take pairs to be the structural form of function designators*, it does not follow that one can apply the function CAR indiscriminately to function designators as arguments. CAR is defined only over those arguments whose *referents* are pairs, not over arguments that *normalise* to pairs. We would have in 2-LISP, for example, the following (this makes use of procedures which will be introduced in the next section, but their 1-LISP analogs will suffice to make the example clear):

```
(CONS 'A 'B)                          ⇒   '(A . B)                    (S4-147)
(CAR (CONS 'A 'B))                    ⇒   'A
(LAMBDA EXPR [X] (+ X 1))             ⇒   (<EXPR> ... )
+                                     ⇒   (<EXPR> ... )
(CAR (LAMBDA EXPR [X] (+ X 1)))       ⇒   <TYPE-ERROR>
(CAR +)                               ⇒   <TYPE-ERROR>
```

In the last two cases, a function defined over pairs was called with an argument that designated a function: hence a type error was recognised. There is in other words no type problem introduced by this choice of normal-form function designator. (We use <EXPR> since the closure that we denote by that abbreviation has no finite lexical notation.)

One might ask how it is noticeable — how one can even tell — that normal-form function designators are pairs. We will provide ways in which it is possible to obtain a designator of the name of an entity — the first of our meta-structural primitives — in section 4.d. As we will explain there, the form "↑<EXP>" — using the NAME function illlustrated in section 4.a.vi — designates a normal-form designator of <EXP>. Using this

explicit mechanism, it will be possible to obtain explicit extensional access to the closure pairs, as illustrated in the following console session (these examples illustrate why NAME is more general than HANDLE):

```
> (CAR (CONS 'A 'B))                                          (S4-148)
> 'A
> CONS
> (<EXPR> ... )
> (CAR CONS)
TYPE-ERROR: CAR, expecting a pair, found the function (<EXPR> ... )
> ↑CONS
> '(<EXPR> ... )
> (CAR ↑CONS)
> '<EXPR>
> ↑(CONS 'A 'B)
> ''(A . B)
> (CAR ↑(CONS 'A 'B))
TYPE-ERROR: CAR, expecting a pair, found the handle '(A . B)
```

Our strict separation of the *reference* relationship and the *normalisation* relationship, in other words, which were conflated by traditional LISPs' notion of evaluation, means not only that we are given an affirmative answer to the question of whether closures should be structures, but also that that answer does not unleash any inelegance or confusion about how pairs and functions can be kept strictly separate.

Two questions remain: what are to be the normal-form designators for the user's world of objects and relationships, and whether atoms are to be normal-form designators at all. Since we know ahead of time nothing about that user's world, we may simply posit that the user may use atoms for normal-form designators for that part of the semantical domain. However such a decision will not much impinge on our investigation, because of the fact that the processor we define is always at least one meta-level away from dealing directly with structures that designate entities in that world. *All* expressions given to the primitive 2-LISP processor, in other words, are of degree at least 2 with respect to the user's world. We provide the space of primitive names (atoms) for the user, in case the user wants to define the user process in a categorically correspondent way with the primitive 2-LISP process, although there is no reason that this would have to be done. For our own purposes, we will assume that no atoms are normal-form designators, and will restrict our attention to $S \cup$ ***FUNCTIONS*** $\cup$ ***ABSTRACTIONS***.

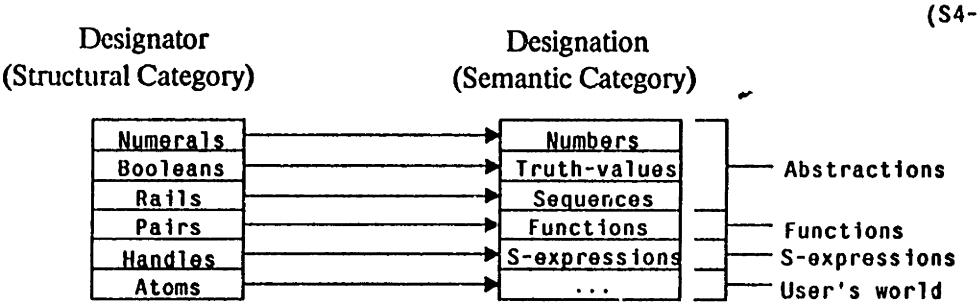
It might seem that providing the atoms for the user's use, as normal-form designators, is a poor offering, since they are content-free (they "contain no information",

in any sense). However two comments argue against this alleged meanness on our part. First, the functions in ***FUNCTIONS*** are defined over all of ***D***, not just over its structural and mathematical components. Second, it is natural in English to use proper names as canonical — even as rigid — designators. Standard names for objects that are *not* proper names are typically formed of functions defined with respect to other proper names. Thus I may have the name *Caitlin* as the standard name of my daughter, and the name *Niagara Falls* as the name of the drop in the river between Lake Ontario and Lake Erie. Suppose she lost a hat on a trip there: I may standardly refer to that hat as *the hat Caitlin lost on our trip to Niagara Falls.* Such a standard name is approximately available in this 2-LISP proposal, since it is constituted of functions defined over atomic and rigid proper names (of course we don't have the definite description operator "the", but the general point remains). Thus the *combination* of functions and atomic names is in fact a more generous allotment than might at first appear.

In addition, of course, there is unlikely to be a serviceable notion of normal-form designator in an actual practical system, even though the search for context-dependent appropriate ways of refering is an important and difficult task. Any real system would in all likelihood impose an entire naming structure, and designation relationship, on top of 2-LISP: our dialect, in fact, would serve only to *implement* such a system, and one of the freedoms that comes from implementing is that one enters an entirely new semantical framework. In such a circumstance, the 2-LISP "data structures" would designate structural elements of the structural field of the implemented architecture — which would presumably be well-defined and straightforwardly denotable. Thus for this reason as well we have no particular cause for worry about the user's domain.

The final arrangement of normal-form designation, then, is summarised in the following diagram:

(S4-149)



4.a.ix. Accessibility

There are two final concerns we must attend to, before looking at the 2-LISP primitive procedures: the locality metric on the field, and graphical notation. In the English describing the six 2-LISP structural categories we made reference to the variety of accessibility relationships for elements of each category. Our mathematical reconstructions, however, did not deal with this aspect of the field — a lack we will now correct, since the formulation of the import of various of the primitive 2-LISP procedures (and even of 2-LISP's Θ) requires reference to this accessibility relationship (CONS, for example, will require such a treatment).

We will define a meta-theoretic function ACCESSIBLE that takes an element S of $\mathcal{S}$, and a field F, onto the set of structures in $\mathcal{S}$ accessible in F from S.

(S4-150)

```
ACCESSIBLE  : [[S × ENVS × FIELDS] → S*]
    ≡ λS ∈ S, E ∈ ENVS, F ∈ FIELDS
        [{HANDLE(S)} ∪
         [case TYPE(S)
            NUMERAL  → {}
            BOOLEAN  → {}
            ATOM     → E(S)
            PAIR     → {CAR(S,F), CDR(S,F)}
            RAIL     → [{T | ∃i [1≤i≤LENGTH(S) [T = NTH(N,S,F)]]} ∪
                        {R | ∃i [1≤i≤LENGTH(S) [R = TAIL(I,S,F)]]}]
            HANDLE   → {HANDLE⁻¹(S)}]]
```

Similarly, ACCESSIBLE* takes an expression onto the transitive closure of ACCESSIBLE: thus, ACCESIBLE*(S) is the set of all elements of S that can be reached in a finite number of local relationships from S.

(S4-151)

```
ACCESSIBLE*  : [[S × ENVS × FIELDS] → S*]
    ≡ λS ∈ S, E ∈ ENVS, F ∈ FIELDS
```

[*the smallest set* $T \subset S$ *such that*
[[ACCESSIBLE(S) $\subset T$] $\wedge$
[$\forall S' \in T$ [ACESSIBLE(S') $\subset T$]]]]

What we then need a name for is the set of all structures that can be reached, in a given context, from structures that can be typed in. Since pairs and rails are created new upon reading, this reduces to those accessible from the numerals, booleans, atoms, and handles. The handles, since they are accessible from their referents, can be ignored (they will be included automatically). Thus we can define:

VISIBLES $\equiv \forall S \in$ [*ATOMS* $\cup$ *NUMERALS* $\cup$ *BOOLEANS*] (S4-152)
the union of ACCESSIBLE*(S)

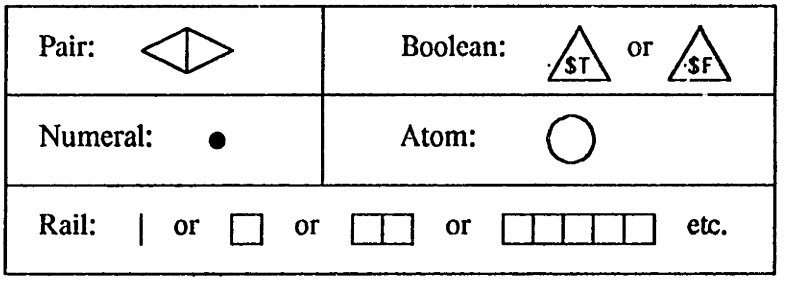
It is this set, for example, that would have to be saved by a garbage collector on a marking or collecting pass. It is this set, in addition, in which pairs and rails notated by parentheses and brackets must *not* fall, by our account of Θ. Though we will not spell out these matters here, some of them will arise when we characterise the full computational significance of structure generating procedures such as CONS.

4.a.x. Graphical Notation

We turn finally to graphical notation. Since we have redefined the structural elements out of which our field is composed, it is clear that the 1-LISP graphical notation we defined in chapter 2 will no longer apply. It will be useful, furthermore, in some of the subsequent discussion to have a notation whose objects correspond one-to-one with the structural field entities they notate. In this section, therefore, we will briefly define an appropriate 2-LISP graphical notation, comprising an ***icon*** type for each of the six structure types, and arrows for the CAR, CDR, FIRST, REST, and PROPERTY relationships.

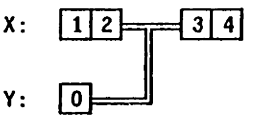
We will generalise the "two-box" icons we used in 1-LISP for pairs, so as to allow any number of boxes, and use it instead to notate rails. Pairs will be demarcated instead with a diamond; the left hand used for the CAR, the right hand side for the CDR. Numerals, atoms, and booleans will be notated with dots, circles, and triangles, but by and large we will simply use their lexical names rather than particular icons. Thus we have the following sample of these five types:

(S4-153)



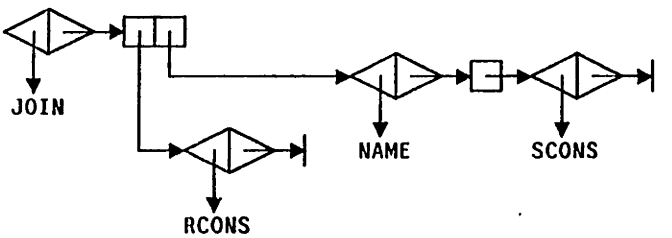
There is one complexity here: since rails can share tails, we need to be able to indicate that graphically (since we have to preserve the one-to-one nature of Θ). Thus between any adjacent boxes in a rail icon we will admit if necessary a double line, connecting at its left hand end with the right hand border of a box, and at its right hand end with the left hand border of the box notating the tail. Thus if X was the rail [1 2 3 4], and Y was [0 3 4], such that the first tail of Y was the same rail as the second tail of X, the following notation would be appropriate:

(S4-154)



In addition, distinct rails can of course have no elements at all (this is what is indicated by the isolated single line at the left of the bottom row of S4-153). Thus, the following notates the structure (JOIN (RCONS) (NAME (SCONS))) (we immediatley begin to use the standard extension of allowing lexical items to replace graphical icons where that is appropriate — particularly for the constants):

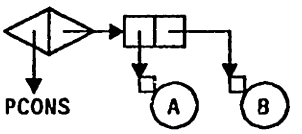
(S4-155)



Finally, we need a notation for handles. Since there is exactly one handle per other structure, we need a convention whereby the handle icon is uniquely associated with the notation for its referent. We will adopt the following protocol: a small box sitting immediately adjacent to and above (usually to the left) of a structure will notate the handle

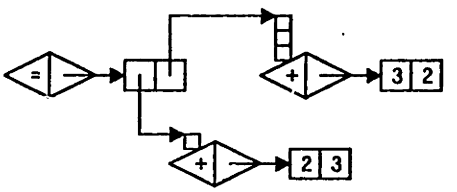
of that structure. Thus the structure lexically notated as `(PCONS 'A 'B)` would have the following graphical image:

(S4-166)



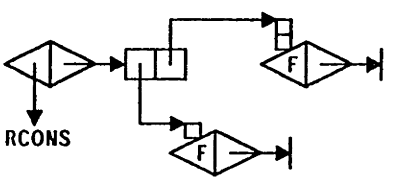
Multiple handles would be notated in the obvious way; thus the following notates the expression `(= '(+ 2 3) '''(+ 3 2))`:

(S4-167)



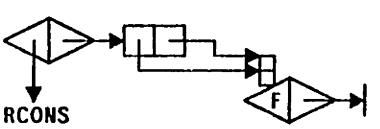
There is, of course, given our protocols on handles, the possibility of using two different handles, one of which is the other's referent (or the other's referent's referent, etc.). If one were to read in the expression `(RCONS '(F) ''(F))`, one would internalise the structure notated as follows:

(S4-168)



However there is another reading of that expression, by which the appropriate graphical notation would be this:

(S4-169)



Though we will not use graphical notation often it will sometimes be crucial in order to demonstrate the token identity of certain circular and shared structures.

4.b. Simple 2-LISP Primitives

We have not yet introduced any of the primitive 2-LISP procedures, although we have introduced more of 2-LISP than would traditionally be the case at such a point, since we defined much of the declarative semantics, and also such formal notions as accessibility, with respect to the field itself, rather than with respect to primitive functions defined over it. Nonetheless, we must now turn to 2-LISP behaviour, in terms of its effect on the structural field laid out in the last section.

There are thirty-two primitive 2-LISP procedures, listed in the table below. The manner in which these procedures are made available is this: in the initial 2-LISP environment (to which we will again meta-theoretically refer using the name E_0) thirty-two atoms are bound to thirty-two *primitively recognised normal-form function designators.* 2-LISP differs from 1-LISP, in other words, in that it is the *closures* that are primitive, rather than their *names.* Though it is convenient to provide standard names for them in the initial environment, these names can be redefined, and other names can be bound to the primitively-recognised closures. It is simpler and more elegant to have all primitives be in normal form (which closures are) rather than having certain context-relative atoms be primitive in some standard initial environment. Just what normal-form designators are structurally like will become clearer in section 4.d after we introduce LAMBDA; first, however, we will simply illustrate their use. Reductions in terms of these primitive closures, in particular, are treated primitively and atomically (i.e., without any observable intermediate states, and, so to speak, in "unit time"), rather than in virtue of any recursive procedural decomposition of their "body" expressions.

The 2-LISP Primitive Procedures (S4-165)

Arithmetic:	+, -, *, /	—	as usual
Typing:	TYPE	—	defined over 6 syntactic and 4 semantic types
Identity:	=	—	s-expressions, truth-values, sequences, numbers
Structural:	PCONS, CAR, CDR	—	to construct and examine pairs
	LENGTH, NTH, TAIL	—	to examine rails and sequences
	RCONS, SCONS, PREP	—	to construct " " "
Modifiers:	RPLACA, RPLACD	—	to modify pairs
	RPLACN, RPLACT	—	to modify rails
I/O:	READ, PRINT, TERPRI	—	as usual
Control:	IF	—	an if-then-else conditional
Naming:	SET, LAMBDA	—	to define, modify, and bind names
Functions:	EXPR, IMPR, MACRO	—	three types of function designator
Semantics:	NAME, REFERENT	—	to mediate between sign and significant
Processor:	NORMALISE, REDUCE	—	primitive access to the processor functions

For each procedure type, three kinds of account are relevant: its *declarative import*, its *procedural consequence*, and an account of how it is computationally tractable (i.e., a computational account of how it can be made to work). In this and the following sections (4.b through 4.c) we will deal with the first two, as embodied in the full significance function Σ; the third will be taken up for the dialect as a whole in section 4.d.vii, when we discuss the 2-LISP meta-circular processor.

4.b.i. Arithmetic Primitives

The four simple "arithmetic" functions (addition, substraction, multiplication, and division) are designated in E_0 by the atoms +, -, *, and /. Thus we have (all of the examples in this section will be given relative to E_0) the following normalisations:

```
(+ 2 3)                ⇒  5                    (S4-166)
(* 10 -4)              ⇒  -40
(/ (* 4 4) (+ 4 4))    ⇒  2
```

Simple as these examples appear, they illustrate a profusion of facts about 2-LISP. We will look in particular, in considerable depth, at the first of these: that the pair (+ 2 3) normalises to the numeral 5.

First, it should be clear that, although the driving behaviour of the 2-LISP processor is one of *normalisation*, not *de-referencing*, these functions (and most other we will examine)

are *declaratively extensional*, in the sense that from a declarative point of view they are defined over the *referents* of their arguments (although procedurally, of course, a different story needs to be told). Thus, although + is a procedure which normalises its arguments, yielding numerals, applications formed in terms of it nonetheless designate the number that is the *sum of the numbers designated by its arguments*, not the number that is the *sum of its normalised arguments*. Similarly, as we will see below, (CAR '(A . B)) normalises its argument, which, being a handle, normalises to itself: '(A . B). Thus CAR, so to speak, "receives" as its intermediate value a handle, not a pair. Nonetheless, (CAR '(A . B)) designates the CAR of the pair designated by that handle — namely, the atom A.

A *normalising processor* and an *extensional semantics* are fully compatible, as of course the λ-calculus and all previous mathematical calculi make manifest. It is this overarching fact that will lead us to a particular definition of EXT for 2-LISP, and will enable us to align EXT and EXPR.

We will consider the (+ 2 3) example in more detail. Structurally, of course, this is a pair, whose CAR is the atom + and whose CDR is a two-element rail, whose first element is the numeral 2, and whose second element is the numeral 3, since "(+ 2 3)" is an abbreviation for "(+ . [2 3])". In E_0 the atom + is bound to a closure — a normal-form function designator — that is circular and primitively recognised (it is the normalisation of the rather un-informative lambda abstraction (LAMBDA EXPR [X Y] (+ X Y))). From the fact that this is an EXPR two things follow: declaratively, it designates an extensional function, and procedurally, it engenders the normalisation of its arguments. We will see the consequences of both of these facts in each of the following two stories, and will then show how in combination they enable us to prove that + satisfies the over-arching normalisation mandate.

First we look at (+ 2 3) declaratively. Φ of the primitive addition closure (in all contexts) is the extensionalisation of the addition function:

$$\forall E \in \textit{ENVS},\ F \in \textit{FIELDS}\ \ [\Phi EF(E_0(\text{"+"})) = EXT(+)] \qquad \text{(S4-167)}$$

where EXT is the 2-LISP extensionalisation function. The version of this meta-theoretic function that we constructed for 1-LISP was complicated by the fact that it had to deal with multiple arguments, but in our present circumstance only the CDR needs to be examined; if that CDR designates a sequence (which it must in order for the whole reduction to be well-

formed semantically) then the computational significance of the CDR will show how that goes. In particular, if the CDR is a rail (the typical case), then the significance of rails set out in S4-105 will play a role. However in general the following definition of EXT will suffice (we start straight away with a definition phrased in terms of the full computational significance Σ):

$$\begin{aligned}\text{EXT} \equiv \lambda G\ .\ [\lambda S.\lambda E.\lambda F.\ & \\ & [\Sigma(S,E,F,\\ & \quad [\lambda\langle S_1,D_1,E_1,F_1\rangle\ .\ G(D_1{}^1,D_1{}^2,\ \ldots\ ,D_1{}^k)]]]\end{aligned} \tag{S4-168}$$

The closure itself is a pair of the following form (by <EXPR>, as noted earlier, we designate the circular closure sketched in S3-200; its full characterisation will be examined in section 4.d.iii):

$$E_0(\text{"+}) = \text{"}(\langle EXPR\rangle\ \underline{E_0}\ [X\ Y]\ (+\ X\ Y)) \tag{S4-169}$$

Finally, the *internalisation* of this closure — the function computed by the processor when processing applications formed in terms of it — is, as we might expect, numeral addition over the results of its arguments (actually over the first and second element of the result of its arguments, since we expect a rail):

$$\begin{aligned}\Delta(E_0(\text{"+})) = \lambda S.\lambda E.\lambda F.\lambda C.\ & \\ & [\Sigma(S,E,F,\\ & \quad [\lambda\langle S_2,D_2,E_2,F_2\rangle\ .\\ & \qquad C(M^{-1}(+(M(\text{NTH}(1,S_2,F_2)),M(\text{NTH}(2,S_2,F_2))),\\ & \qquad\ E_2,F_2)]]\end{aligned} \tag{S4-170}$$

This is sufficient characterisation to prove anything we need to prove about (binary) addition, but before turning to an example we should straightaway define some meta-theoretic machinery that will enable us to say what we have just said much more compactly. In particular, note that the internalisation of the plus closure contains some complexity having to do with the normalisation of its arguments; it would be convenient if, instead of writing S4-170, we would more simply say (since this has inherently to do with the fact that + is an EXPR):

$$\Delta(E_0(\text{"+})) = \text{EXPR}(\lambda\langle N_1,N_2\rangle\ .\ M^{-1}(+(M(N_1),M(N_2)))) \tag{S4-171}$$

To translate this into English, this simply states that the internalisation of the (primitive) addition closure is EXPR of numeral addition. We had no need to talk of the full significance of the arguments, continuations, or the rest.

What this requires is a suitable definition of EXPR, which is easy to define:

$$\texttt{EXPR} \equiv \lambda G\ .\ [\lambda S.\lambda E.\lambda F.\lambda C.\ [\Sigma\langle S,E,F,\ [\lambda\langle S_1,D_1,E_1,F_1\rangle\ .\ C(G\langle NTH(1,S_1,F_1),NTH(2,S_1,F_1),\ \ldots\ ,NTH(k,S_1,F_1)\rangle,\ E_1,F_1)]]] \qquad \text{(S4-172)}$$

Thus S4-171 can be taken as equivalent to S4-170. To review, we can then set out the full computational significance of the atom + in the initial environment. This takes two parts (as we saw in the last chapter): its Σ-characterisation, and the additional internalisation of its local procedural significance:

$$\Sigma(E_0(\text{"+})) = \lambda E.\lambda F.\lambda C.\ [C(\text{"}(\langle EXPR\rangle\ \underline{E_0}\ [X\ Y]\ (+\ X\ Y)),\ EXT(+),\ E,F)] \qquad \text{(S4-173)}$$

and

$$\Delta(E_0(\text{"+})) = EXPR(\lambda\langle N_1,N_2\rangle\ .\ M^{-1}(+(M(N_1),M(N_2)))) \qquad \text{(S4-174)}$$

Finally, we collapse these two into a single notion of being ***simple***. There are two salient facts about the previous two equations. First, all the signified computations are side-effect free. Second, there are three pieces of information beyond that, that need to be stated: the form of the primitive closure, the designated function, and the internal function. Therefore we can define the following meta-theoretic function:

$$\texttt{SIMPLE} \equiv \lambda\langle L,G_1,G_2\rangle\ .\ [[E_0(L) = \ulcorner\text{"}(\langle EXPR\rangle\ \underline{E_0}\ [V_1\ V_2\ \ldots\ V_k]\ (\underline{L}\ V_1\ V_2\ \ldots\ V_k))\urcorner] \wedge [\Sigma(E_0(L)) = \lambda E.\lambda F.\lambda C\ .\ C(E_0(L),EXT(G_1),E,F)] \wedge [\Delta[E_0(L)] = EXPR(G_2)]] \qquad \text{(S4-175)}$$

The variables L, G_1, and G_2 in this definition are intended to be bound to an atomic label, a function (to be extensionalised), and another function that is the internalisation of the primitive closure. Thus we can assert:

$$\texttt{SIMPLE}(\text{"+},+,[\lambda\langle N_1,N_2\rangle\ .\ M^{-1}(+(M(N_1),M(N_2)))]) \qquad \text{(S4-176)}$$

This single formula encodes all we need to say about addition; thus we can use it to completely characterise the semantics of the other three arithmetic operators:

$$\texttt{SIMPLE}(\text{"*},*,[\lambda\langle N_1,N_2\rangle\ .\ M^{-1}(*(M(N_1),M(N_2)))]) \qquad \text{(S4-177)}$$
$$\texttt{SIMPLE}(\text{"-},-,[\lambda\langle N_1,N_2\rangle\ .\ M^{-1}(-(M(N_1),M(N_2)))])$$
$$\texttt{SIMPLE}(\text{"/},/,[\lambda\langle N_1,N_2\rangle\ .\ M^{-1}(/(M(N_1),M(N_2)))])$$

(Strictly, of course, SIMPLE would need to know the number of arguments (K) of the functions in question. This could be repaired either by passing that number to SIMPLE as a fourth argument, or by using an indefinite number of different versions of SIMPLE; in the latter case S4-175 could be taken as a *definition schema*, rather than as a definition itself. We will not worry about this here, as the intent is clear. The problem, furthermore, as the reader will have noticed, is not restricted to SIMPLE: we would need special versions of EXT, EXPR, and so forth. But this could all be taken care of without interest.)

In order to see this characterisation at work, we will look in full at the significance of the term (+ 2 3). We repeat here, for reference, equations S4-21, S4-29, S4-38, and S4-105 that give the declarative import of numerals, atoms, pairs, and rails, respectively:

```
∀N ∈ NUMERALS, E ∈ ENVS, F ∈ FIELDS, C ∈ CONTS                    (S4-178)
    [Σ(N, E, F, C) = C(N, M(N), E, F)]

∀A ∈ ATOMS, E ∈ ENVS, F ∈ FIELDS, C ∈ CONTS                       (S4-179)
    [Σ(A, E, F, C) = C(E(A), ΦEF(E(A)), E, F)]

∀P ∈ PAIRS, E ∈ ENVS, F ∈ FIELDS, C ∈ CONTS                       (S4-180)
  Σ(P,E,F,C) = Σ(F¹(P),E,F,
    [λ<S₁,D₁,E₁,F₁> .
       [(ΔS₁)(F₁²(P),E₁,F₁,
              [λ<S₂,E₂,F₂> .
                    C(S₂,[D₁(F₁²(P),E₁,F₁)],E₂,F₂)])]])

∀R ∈ RAILS, E ∈ ENVS, F ∈ FIELDS, C ∈ CONTS                       (S4-181)
  [Σ(R,E,F,C) =
     if [∀i 1≤i≤LENGTH(R,F) [NORMAL-FORM(NTH(i,R,F))]
       then C(R,D₀,E,F)
       elseif [NTH(1,R,F) = ⊥]
              then C("[],<>,E,F) where "[] is inaccessible in F
              else Σ(NTH(1,R,F),E,F,
                     [λ<S₁,D₁,E₁,F₁> .
                        Σ(REST(R,F),E₁,F₁,
                          [λ<R₂,D₂,E₂,F₂> . C(S,D,E₂,F₃)])])
     where S ∈ RAILS and D ∈ SEQUENCES and D₀ ∈ SEQUENCES;
           NTH(1,S,F₃) = S₁;
           REST(S,F₃) = R₂;
           F₃ = F₂ otherwise;
           LENGTH(R,F) = LENGTH(D₀) = LENGTH(D);
           D¹ = D₁;
           ∀i 1≤i≤LENGTH(D₂) [ D^(i+1) = D₂^i ];
           ∀i 1≤i≤LENGTH(D₀) [ D₀^i = ΦEF(NTH(i,R,F)) ]
```

In terms of all of these, we can prove that (+ 2 3) designates the number five, and returns the numeral 5, in E_0. We will look, in particular, at the meta-theoretic term:

$$\Sigma(\text{"}(+\ 2\ 3),E_0,F_0,ID) \qquad \text{(S4-182)}$$

First we apply S4-181 since (+ 2 3) is a pair:

$$\Sigma(\text{"}(+\ 2\ 3),E_0,F_0,ID) \qquad \text{(S4-183)}$$
$$= \Sigma(F_0^1(\text{"}(+\ 2\ 3)),E_0,F_0,$$
$$[\lambda\langle S_1,D_1,E_1,F_1\rangle\ .$$
$$[(\Delta S_1)(F_1^2(\text{"}(+\ 2\ 3)),E_1,F_1,$$
$$[\lambda\langle S_2,E_2,F_2\rangle . ID(S_2,[D_1(F_1^2(\text{"}(+\ 2\ 3)),E_1,F_1)],E_2,F_2)])]])$$

Performing the CAR on F_0 to extract the function designator, and ridding ourselves of the inconsequential ID, leads to:

$$= \Sigma(\text{"}+,E_0,F_0, \qquad \text{(S4-184)}$$
$$[\lambda\langle S_1,D_1,E_1,F_1\rangle\ .$$
$$[(\Delta S_1)(F_1^2(\text{"}(+\ 2\ 3)),E_1,F_1,$$
$$[\lambda\langle S_2,E_2,F_2\rangle\ .\ \langle S_2,[D_1(F_1^2(\text{"}(+\ 2\ 3)),E_1,F_1)],E_2,F_2\rangle])]])$$

The term "+ is an atom; thus S4-179 applies:

$$= ([\lambda\langle S_1,D_1,E_1,F_1\rangle\ . \qquad \text{(S4-185)}$$
$$[(\Delta S_1)(F_1^2(\text{"}(+\ 2\ 3)),E_1,F_1,$$
$$[\lambda\langle S_2,E_2,F_2\rangle\ .\ \langle S_2,[D_1(F_1^2(\text{"}(+\ 2\ 3)),E_1,F_1)],E_2,F_2\rangle])]]$$
$$\langle E_0(\text{"}+),\Phi E_0F_0(E_0(\text{"}+)),E_0,F_0\rangle)$$

We are now ready for some addition-specific reductions. In particular, we insert the E_0 binding of the atom +, and its designation in that context:

$$= ([\lambda\langle S_1,D_1,E_1,F_1\rangle\ . \qquad \text{(S4-186)}$$
$$[(\Delta S_1)(F_1^2(\text{"}(+\ 2\ 3)),E_1,F_1,$$
$$[\lambda\langle S_2,E_2,F_2\rangle\ .\ \langle S_2,[D_1(F_1^2(\text{"}(+\ 2\ 3)),E_1,F_1)],E_2,F_2\rangle])]]$$
$$\langle\text{"}(\langle EXPR\rangle\ \underline{E_0}\ [X\ Y]\ (+\ X\ Y)),EXT(+),E_0,F_0\rangle)$$

Expanding next the extensionalisation of the (meta-theoretic) addition function, we get:

$$= ([\lambda\langle S_1,D_1,E_1,F_1\rangle\ . \qquad \text{(S4-187)}$$
$$[(\Delta S_1)(F_1^2(\text{"}(+\ 2\ 3)),E_1,F_1,$$
$$[\lambda\langle S_2,E_2,F_2\rangle\ .\ \langle S_2,[D_1(F_1^2(\text{"}(+\ 2\ 3)),E_1,F_1)],E_2,F_2\rangle])]]$$
$$\langle\text{"}(\langle EXPR\rangle\ \underline{E_0}\ [X\ Y]\ (+\ X\ Y)),$$
$$([\lambda G\ .\ [\lambda S.\lambda E.\lambda F.$$
$$[\Sigma(S,E,F,\ [\lambda\langle S_1,D_1,E_1,F_1\rangle\ .\ G(D_1^1,D_1^2,\ \ldots\ ,D_1^k)]]]]$$
$$+),$$
$$E_0,$$
$$F_0\rangle)$$

This extensionalisation can be reduced:

$$= ([\lambda\langle S_1,D_1,E_1,F_1\rangle\ . \qquad \text{(S4-188)}$$
$$[(\Delta S_1)(F_1^2(\text{"}(+\ 2\ 3)),E_1,F_1,$$
$$[\lambda\langle S_2,E_2,F_2\rangle\ .\ \langle S_2,[D_1(F_1^2(\text{"}(+\ 2\ 3)),E_1,F_1)],E_2,F_2\rangle])]]$$
$$\langle\text{"}(\langle EXPR\rangle\ \underline{E_0}\ [X\ Y]\ (+\ X\ Y)),$$

```
[λS.λE.λF.
      [Σ(S,E,F, [λ<S1,D1,E1,F1> . +(D1^1,D1^2, ... ,D1^k)]]],
E0,
F0>)
```

There are no further reductions applicable to the four arguments to the continuation; we can therefore reduce it (not, of course, that the reduction order matters — this, after all, is the λ-calculus — but applicative order seems the most natural way to proceed):

```
= [(Δ["(<EXPR> E0 [X Y] (+ X Y))])                                    (S4-189)
   <F0^2("(+ 2 3)),E0,F0,
    [λ<S2,E2,F2> .
        <S2,
         ([λS.λE.λF . [Σ(S,E,F, [λ<S1,D1,E1,F1> . +(D1^1,D1^2, ... ,D1^k)]]]
          <F0^2("(+ 2 3)),E0,F0>),
         E2,
         F2>]>]
```

We can reduce the innermost application formed in terms of the extensionalised addition function, after performing that straightforward CDR on F_0 (reducing, in other words, F_0^2("(+ 2 3)) to "[2 3]):

```
= [(Δ["(<EXPR> E0 [X Y] (+ X Y))])                                    (S4-190)
   <F0^2("(+ 2 3)),E0,F0,
    [λ<S2,E2,F2> .
        <S2,
         [Σ("[2 3],E0,F0, [λ<S1,D1,E1,F1> . +(D1^1,D1^2, ... ,D1^k)]]
         E2,
         F2>)]>]
```

We turn now to the full significance of the expression [2 3] in the initial context. Though we do this in full here, it will (as was the case in the examples of last chapter) arise again below, where we will carry over this formulation intact. The term in question, of course, is a rail; thus a use of S4-181 is indicated. In the present case all elements of the rail are in normal form; thus the rail itself is returned rather straightforwardly. We have indicated straight away that all of the elements of the designated sequence are numbers; this is implied by the significance of numerals manifested in S4-178:

```
= [(Δ["(<EXPR> F0 [X Y] (+ X Y))])                                    (S4-191)
   <F0^2("(+ 2 3)),E0,F0,
    [λ<S2,E2,F2> .
        <S2,
         ([λ<S1,D1,E1,F1> . +(D1^1,D1^2)] <"[2 3],<2,3>,E0,F0>)
         E2,
         F2>]>]
```

A simple reduction leads to:

```
= [(Δ["(<EXPR> E0 [X Y] (+ X Y))])                                    (S4-192)
   <F0^2("(+ 2 3)),E0,F0,
    [λ<S2,E2,F2> . <S2,+(2,3),E2,F2>]>]
```

Performing the addition, we have proved that `(+ 2 3)` designates the number 5. We can also execute the outstanding CDR to obtain the arguments to the *internal* addition function:

```
= [(Δ["(<EXPR> E0 [X Y] (+ X Y))])                                    (S4-193)
   <"[2 3],E0,F0,[λ<S2,E2,F2> . <S2,5,E2,F2>]>]
```

Next we need to explore the internalised function engendered by the primitive addition closure. This was set forth in S4-174; it leads to:

```
= ([λS.λE.λF.λC.                                                      (S4-194)
       [Σ(S,E,F,
          [λ<S2,D2,E2,F2> .
                C(M^-1(+(M(NTH(1,F2,S2)),M(NTH(2,F2,S2))),E2,F2)]]]
   <"[2 3],E0,F0,[λ<S2,E2,F2> . <S2,5,E2,F2>]>)
```

In preparation for the next reduction, we need to perform an α-reduction to avoid potential variable collisions:

```
= ([λS.λE.λF.λC.                                                      (S4-195)
       [Σ(S,E,F,
          [λ<S2,D2,E2,F2> .
                C(M^-1(+(M(NTH(1,F2,S2)),M(NTH(2,F2,S2))),E2,F2)]]]
   <"[2 3],E0,F0,[λ<S3,E3,F3> . <S3,5,E3,F3>]>)
```

Then applying the arguments:

```
= [Σ("[2 3],E0,F0,                                                    (S4-196)
    [λ<S2,D2,E2,F2> .
          ([λ<S3,E3,F3> . <S3,5,E3,F3>]
           <M^-1(+(M(NTH(1,F2,S2)),M(NTH(2,F2,S2))),E2,F2>)]]
```

Once again we need the full significance of the normal-form rail `[2 3]`; once again it is simple:

```
= ([λ<S2,D2,E2,F2> .                                                  (S4-197)
       ([λ<S3,E3,F3> . <S3,5,E3,F3>]
        <M^-1(+(M(NTH(1,F2,S2)),M(NTH(2,F2,S2))),E2,F2>)]
   <"[2 3],<2,3>,E0,F0>)
```

Applying this, we are set to perform the numeral addition (note that this time the abstract sequence `<2,3>` is ignored):

$$= ([\lambda\langle S_3,E_3,F_3\rangle\ .\ \langle S_3,5,E_3,F_3\rangle] \quad \text{(S4-198)}$$
$$\langle M^{-1}(+(M(NTH(1,F_0,"[2\ 3]")),M(NTH(2,F_0,"[2\ 3]"))),E_0,F_0\rangle)$$

Expanding the NTHS:

$$= ([\lambda\langle S_3,E_3,F_3\rangle\ .\ \langle S_3,5,E_3,F_3\rangle] \quad \text{(S4-199)}$$
$$\langle M^{-1}(+(M("2),M("3)),E_0,F_0\rangle)$$

The numeral addition is simple:

$$= ([\lambda\langle S_3,E_3,F_3\rangle\ .\ \langle S_3,5,E_3,F_3\rangle]\ \langle M^{-1}(+(2,3),E_0,F_0\rangle) \quad \text{(S4-200)}$$

$$= ([\lambda\langle S_3,E_3,F_3\rangle\ .\ \langle S_3,5,E_3,F_3\rangle]\ \langle M^{-1}(5),E_0,F_0\rangle) \quad \text{(S4-201)}$$

$$= ([\lambda\langle S_3,E_3,F_3\rangle\ .\ \langle S_3,5,E_3,F_3\rangle]\ \langle "5,E_0,F_0\rangle) \quad \text{(S4-202)}$$

Finally, the top level continuation puts together the designation (the number five) and the result (the numeral 5), for a full significance of:

$$= \langle "5,5,E_0,F_0\rangle \quad \text{(S4-203)}$$

What then have we done? A number of things. First, we have shown, in a proof that in many ways resembles the example comprising section 3.e.iv, how a complete derivation of the full significance of an expression yields both its designation and its result, as well as manifesting any side-effects that may have occurred during its processing. Like that example, this case involved no side effects; unlike that case, however, we have shown how the result and the designation need not be the same. In particular, whereas the 1-LISP expression (CAR '(A B C)) designated what it returned, the 2-LISP term (+ 2 3) designated an abstract number, but returned the numeral.

Thus the local procedural import of (+ 2 3) is the numeral that designates the declarative import. Because of this fact, and because of the ancillary fact that numerals are normal form, we have proved the following very particular instance of the normalisation theorem:

$$[[\Phi E_0F_0(\Psi E_0F_0("(+\ 2\ 3))) = \Phi E_0F_0("(+\ 2\ 3))]\ \wedge \quad \text{(S4-204)}$$
$$[NORMAL\text{-}FORM(\Psi E_0F_0("(+\ 2\ 3)))]]$$

We have looked carefully at (+ 2 3) from two points of view: declarative and procedural semantics. In order to complete the analysis, we will very briefly examine it from the third, computational, standpoint, inquiring as to how it is actually manipulated by the formal processor.

First, when the pair (+ 2 3) is normalised in E_0, the CAR of the pair is normalised in E_0, as mandated by S4-180. Since that CAR is an atom, the binding of + is retrieved (mandated by S4-173), yielding the closure (<EXPR> $\underline{E_0}$ [X Y] (+ X Y)). Since that closure is discernibly an EXPR, the CDR of the original pair is normalised next, still in E_0. The CDR in this case is the rail [2 3]; when a rail is normalised, the processor first determines whether the rail is a normal-form designator already — a condition true just in case the elements are themselves in normal form. In our case there are two elements, both of which are numerals; hence the rail is in normal-form, and is therefore "returned" as the normalisation of the CDR.

The closure and the normal-form rail are then reduced — primitively, in this case, since the closure is a primitive closure. It is part of the definition of the 2-LISP processor that the appropriate numeral addition function is effected in a single step. We will see how "answers" are returned in terms of continuation in due course; for the time being we can merely see that, in our particular case, the numeral 5 is returned as the normalisation of the original expression (+ 2 3).

More details on how the 2-LISP processor may be embodied will, as we mentioned earlier, be taken up in section 4.d.vii. However two important points should be made here. First, because this is a *computational* system, there is no sense in which the *designation* of any term is produced, examined, looked at, or anything else, from the point of view of the processor. *It* in no way knows that (+ 2 3) designates five; nor does it know that [2 3] designates the abstract sequence of numbers. Nor, for that matter, does it care. The entire machinery in our meta-language dealing with declarative import merely assures us that our pre-theoretic attribution of meaning to 2-LISP structures remains alligned with what the processor does. Computation, from beginning to end, is formal.

One other general comment needs to be made before we look at other primitive procedures, in a less mathematical way. We have shown that four of the six 2-LISP structures satisfy the normalisation mandate: the booleans, the numerals, the handles, and the rails (providing, in the last case, that their elements satisfy it). We have shown how a particular example of a pair satisfied the theorem, but we have of course not shown that pairs do in general. Nor have we shown that *bindings* are in normal-form, which would be required in order to show that *atoms* satisfy the theorem in all contexts. We will, as promised, not do this: the basic structure of such a proof, however, is exhibited in the

structure of the examples we have given. The strategy would be similar to that suggested at the end of the previous chapter, where we discussed proving the corresponding evaluation theorem for 1-LISP: we would show that if all arguments to a pair satisfied the theorem, and if the function were *standard*, then the pair itself would satisfy it. We would then show that all primitive procedures were standard, and that all procedures composable and definable within the dialect were standard if their consituents were standard. But the mathematics has been sufficient for our present purposes. In what follows we will ease up on formalism, in order better to convey the subtlety of the *particular* properties of the procedures to be introduced. The normalisation theorem mandates a *general* semantical cast to be honored by all 2-LISP expressions; these few simple examples have shown how this general property can be straightforwardly embodied in an approximately familiar dialect.

We conclude this section with a final comment about 2-LISP arithmetic. Since the examples of the use of the 2-LISP arithmetic functions are so simple, there is a tendency to think that there are no discernable surface differences from the behaviour of 1-LISP. That this is not so, is easily demonstrable by making some errors. Consider for example the following example of 1-LISP evaluation:

```
> (+ 3 '4)              ; This is 1-LISP                    (S4-205)
> 7
```

This "works", of course, because in 1-LISP numerals evaluate to themselves, whereas quoted numerals evaluate to numerals as well. In 2-LISP, on the other hand, we would encounter the following:

```
> (+ 3 '4)              ; This is 2-LISP                    (S4-206)
TYPE-ERROR: +, expecting a number, found the numeral '4
```

This is of course correct; the expression "'4" is a handle, designating a numeral, and addition is defined over numbers, not over numerals. If this example were analysed semantically in the manner of our long example above, it would emerge in the line of the derivation corresponding to S4-201 that the real addition function would be applied to the abstract sequence <3,"4>, which of course is inadmissable.

2-LISP, it may be said, is semantically strict. For such very trivial examples as these this strictness might seem an inconvenience or even a mis-feature. When we turn to questions of reflection, however, we will see that the ability to rely on the semantical

strictness — in particular, on the fact that normalisation process never crosses semantic meta-levels — is a great boon, engendering great flexibility.

4.b.ii. Selectors on Pairs

We turn next to the simplest of the 2-LISP primitives that allow one to examine structures: CAR and CDR. These are extensional functions; thus (CAR X) will designate the CAR of the structure designated by X in the context of use. As was true in 1-LISP, however, we cannot define them in terms of EXT or SIMPLE, because we need to use the field passed in as an argument. What we aim for is something that captures the indended meaning of the following formula — i.e., the appropriate modification of this that ensures that F is bound (we will focus on CAR; CDR is entirely parallel):

```
SIMPLE("CAR,                                                    (S4-210)
       [λX.CAR(X,F)],
       [λX.HANDLE(CAR(HANDLE⁻¹(X),F))]
```

For example, (CAR '(A . B)) will designate the CAR of the pair designated by the argument, which is a handle that designates the pair (A . B). The whole expression, therefore, designates the atom A. The expression will therefore *normalise* to the normal-form designator of that atom, which is the handle 'A. In other words:

```
(CAR '(A . B))      ⇒    'A                                     (S4-211)
```

The semantical equations that will engender this behaviour are straightforward. First the full significance:

```
Σ(E₀("CAR)) = [λE.λF.λC.                                        (S4-212)
     C("(EXPR E₀ [X] (CAR X)),
       [λS₁.λE₁.λF₁.
          Σ(S₁,E₁,F₁,[λ<S₂,D₂,E₂,F₂> . CAR(D₂,F₂)])]
       E,
       F)]
```

Second, the internalisation of the primitive CAR closure:

```
Δ[E₀("CAR)] = λS₁.λE₁.λF₁.λC.                                   (S4-213)
                 [Σ(S₁,E₁,F₁,
                    [λ<S₃,D₃,E₃,F₃> .
                          C(HANDLE(CAR(HANDLE⁻¹(NTH(1,S₃,F₃)))),E₃,F₃)])]
```

In English, what these two together imply is that in any context, the primitive CAR closure (the binding of CAR in E_0) signifies a function that normalises its argument (S_1). Both the declarative and procedural treatments, as usual, are formulated in terms of the full significance of that argument: what it returns is called both S_2 and S_3; what it designates is

called both D_2 and D_3. It follows from the equations that any application formed in terms of the CAR function will designate the CAR of D_2 in the field that is returned; similarly (from the internalised function) we can see that it will *return* a handle designating that CAR. We know this because it returns a handle of the CAR of the referent of S_3, and, because of the normalisation theorem, we know that S_3 designates D_3, and we have just pointed out that D_3 and D_2 are the same entity.

We need not present a complete derivation of an example, as it would be analogous in structure to the one given in the previous section. Of more interest is the following proof that CAR is ***standard***, in the sense of that word first introduced in section 3.e.v. The appropriate definition of that term for 2-LISP (for a normalising language, in particular) is the following:

```
STANDARD  : [ S → {Truth, Falsity} ]                                    (S4-214)
   ≡ λS ∈ S .
       [∀P ∈ PAIRS, E ∈ ENVS, F ∈ FIELDS
            [[CAR(P,F) = S] ⊃
            [[ΦEF(ΨEF(P)) = ΦEF(P)] ∧ [NORMAL-FORM(ΨEF(P))]]]
```

What we wish to prove is the following:

```
STANDARD(E0("CAR))                                                      (S4-216)
```

We will need the following definitions of Φ and Ψ in terms of Σ, from S3-130:

$$\Psi \equiv \lambda E.\lambda F.\lambda S\ .\ [\Sigma(S,E,F,[\lambda\langle X_1,X_2,X_3,X_4\rangle\ .\ X_1])] \qquad \text{(S4-216)}$$
$$\Phi \equiv \lambda E.\lambda F.\lambda S\ .\ [\Sigma(S,E,F,[\lambda\langle X_1,X_2,X_3,X_4\rangle\ .\ X_2])]$$

The first move in our proof is a recasting of the definition of STANDARD; although S4-214 best manifests the intent of the predicate, the following obviously equivalent formulation is evidently easier to prove:

```
STANDARD  : [ S → {Truth, Falsity} ]                                    (S4-217)
   ≡ λS ∈ S .
       [∀P ∈ PAIRS, E ∈ ENVS, F ∈ FIELDS
            [[CAR(P,F) = S] ⊃
            [Σ(P,E,F,
                 [λ⟨R1,D1,E1,F1⟩ .
                    Σ(R1,E1,F1,
                      [λ⟨R2,D2,E2,F2⟩ .
                         [[D1 = D2] ∧ [NORMAL-FORM(R1)]]])]]]]
```

There are a variety of things to note straightaway about this formulation. First, we use E_1 and F_1 to establish the designation D_2 of R_1, rather than E and F; this was mentioned earlier as being the more proper approach. Secondly, it follows, if we can prove the NORMAL-

FORM(R_1) part, that:

$$[[E_1 = E_2] \wedge [F_1 = F_2]] \qquad \text{(S4-218)}$$

since all normal-form expressions must be side effect free.

The first step is a statement of the full significance of pairs (this is S4-38):

```
∀E ∈ ENVS, F ∈ FIELDS, C ∈ CONTS, P ∈ PAIRS                          (S4-219)
   Σ(P,E,F,C) =
       Σ(CAR(P,F),E,F,
         [λ<S1,D1,E1,F1> .
              [(ΔS1)(CDR(P,F1),E1,F1,
                    [λ<S2,E2,F2> . C(S2,[D1(CDR(P,F1),E1,F1)],E2,F2)])]])
```

If we particularise this to a situation in which the CAR of the pair in F is E_0("CAR) we get:

```
∀E ∈ ENVS, F ∈ FIELDS, C ∈ CONTS, P ∈ PAIRS                          (S4-220)
   [[CAR(P,F) = E0("CAR)] ⊃
    [Σ(P,E,F,C) =
        Σ(E0("CAR),E,F,
          [λ<S1,D1,E1,F1> .
               [(ΔS1)(CDR(P,F1),E1,F1,
                     [λ<S2,E2,F2> . C(S2,[D1(CDR(P,F1),E1,F1)],E2,F2)])]])
```

Now, however, the second part of this can be expanded, by applying S4-212:

```
∀E ∈ ENVS, F ∈ FIELDS, C ∈ CONTS, P ∈ PAIRS                          (S4-221)
   [[CAR(P,F) = E0("CAR)] ⊃
    [Σ(P,E,F,C) =
         [([λ<S1,D1,E1,F1> .
               [(ΔS1)(CDR(P,F1),E1,F1,
                     [λ<S2,E2,F2> . C(S2,[D1(CDR(P,F1),E1,F1)],E2,F2)])]])
          ("(EXPR E0 [X] (CAR X)),
           [λS1.λE1.λF1.
              Σ(S1,E1,F1,[λ<S2,D2,E2,F2> . CAR(D2,F2)])]
           E,
           F)]
```

We can reduce this:

```
∀E ∈ ENVS, F ∈ FIELDS, C ∈ CONTS, P ∈ PAIRS                          (S4-222)
   [[CAR(P,F) = E0("CAR)] ⊃
    [Σ(P,E,F,C) =
         [(Δ["(EXPR E0 [X] (CAR X))])
          <CDR(P,F),E,F,
           [λ<S2,E2,F2> .
               C(S2,
                 ([λS1.λE1.λF1.
                      Σ(S1,E1,F1,[λ<S2,D2,E2,F2> . CAR(D2,F2)]
                   <CDR(P,F),E,F>),
                  E2,
                  F2)]>]]]
```

Now we may substitute the internalised CAR function from S4-213:

```
∀E ∈ ENVS, F ∈ FIELDS, C ∈ CONTS, P ∈ PAIRS                          (S4-223)
   [[CAR(P,F) = E0("CAR)] ⊃
    [Σ(P,E,F,C) =
         ([λS1.λE1.λF1.λC.
              [Σ(S1,E1,F1,
                 [λ<S3,D3,E3,F3> .
                       C(HANDLE(CAR(HANDLE⁻¹(NTH(1,S3,F3)))),E3,F3)])]]
          <CDR(P,F),E,F,
           [λ<S2,E2,F2> .
               C(S2,
                 ([λS1.λE1.λF1.
                      Σ(S1,E1,F1,[λ<S2,D2,E2,F2> . CAR(D2,F2)]
                  <CDR(P,F),E,F>),
                 E2,
                 F2)]>]]]
```

This too can be reduced:

```
∀E ∈ ENVS, F ∈ FIELDS, C ∈ CONTS, P ∈ PAIRS                          (S4-224)
   [[CAR(P,F) = E0("CAR)] ⊃
    [Σ(P,E,F,C) =
         [Σ(CDR(P,F),E,F,
                 [λ<S3,D3,E3,F3> .
                      ([λ<S2,E2,F2> .
                           C(S2,
                             ([λS1.λE1.λF1.
                                 Σ(S1,E1,F1,[λ<S2,D2,E2,F2> . CAR(D2,F2)]
                              <CDR(P,F),E,F>),
                             E2,
                             F2)]
                       <HANDLE(CAR(HANDLE⁻¹(NTH(1,S3,F3)))),E3,F3>)])]]]
```

And again:

```
∀E ∈ ENVS, F ∈ FIELDS, C ∈ CONTS, P ∈ PAIRS                          (S4-225)
   [[CAR(P,F) = E0("CAR)] ⊃
    [Σ(P,E,F,C) =
         [Σ(CDR(P,F),E,F,
            [λ<S3,D3,E3,F3> .
                 C(HANDLE(CAR(HANDLE⁻¹(NTH(1,S3,F3)))),
                   ([λS1.λE1.λF1.
                        Σ(S1,E1,F1,[λ<S2,D2,E2,F2> . CAR(D2,F2)])]
                    <CDR(P,F),E,F>),
                   E3,
                   F3)])]]]
```

And again:

```
∀E ∈ ENVS, F ∈ FIELDS, C ∈ CONTS, P ∈ PAIRS                          (S4-226)
   [[CAR(P,F) = E0("CAR)] ⊃
    [Σ(P,E,F,C) =
         [Σ(CDR(P,F),E,F,
                 [λ<S3,D3,E3,F3> .
```

```
C(HANDLE(CAR(HANDLE⁻¹(NTH(1,S₃,F₃)))),
  Σ(CDR(P,F),E,F,[λ<S₂,D₂,E₂,F₂> . CAR(D₂,F₂)],
  E₃,
  F₃)])]]]
```

This is as far as we can reduce without knowing about the CDR; it is a good time as well to review what this says. It is exactly what we would expect the full computational significance of any application in terms of the CAR function will be the following: it will normalise the CDR of the application, since CAR is an EXPR (this is the fourth line of S4-226), which will return a result (S_3), a designation (D_3), and a revised context (E_3 and F_3). The significance of the whole application will be the four-tuple of the handle designating the CAR of the result, the actual CAR of the result, and the context as received from the arguments (i.e., the application of the CAR procedure itself doesn't further modify the context).

We are almost done with our proof. Two steps remain. First, we can, using universal instantiation, construct a more particular version of S4-226, with a particular continuation C: namely, the continuation

```
[λ<R₁,D₁,E₁,F₁> . <R₁,D₁,E₁,F₁>]                                    (S4-227)
```

In particular, we get the following instance:

```
∀E ∈ ENVS, F ∈ FIELDS, P ∈ PAIRS                                    (S4-228)
   [[CAR(P,F) = E₀("CAR)] ⊃
    [Σ(P,E,F,[λ<R₁,D₁,E₁,F₁> . <R₁,D₁,E₁,F₁>])
       = [Σ(CDR(P,F),E,F,
              [λ<S₃,D₃,E₃,F₃> .
                   ([λ<R₁,D₁,E₁,F₁> . <R₁,D₁,E₁,F₁>]
                    <HANDLE(CAR(HANDLE⁻¹(NTH(1,S₃,F₃)))),
                     Σ(CDR(P,F),E,F,[λ<S₂,D₂,E₂,F₂> . CAR(D₂,F₂)],
                     E₃,
                     F₃>)])]]
```

However, since this is essentially a null continuation, the second part of this can be reduced:

```
∀E ∈ ENVS, F ∈ FIELDS, P ∈ PAIRS                                    (S4-229)
   [[CAR(P,F) = E₀("CAR)] ⊃
    [Σ(P,E,F,[λ<R₁,D₁,E₁,F₁> . <R₁,D₁,E₁,F₁>])
       = [Σ(CDR(P,F),E,F,
              [λ<S₃,D₃,E₃,F₃> .
                   <HANDLE(CAR(HANDLE⁻¹(NTH(1,S₃,F₃)))),
                    Σ(CDR(P,F),E,F,[λ<S₂,D₂,E₂,F₂> . CAR(D₂,F₂)],
                    E₃,
                    F₃>)])]]
```

The other component of the proof is the inductive part; we are allowed to assume that the normalisation theorem holds for the *arguments* to CAR; we are allowed to assume, in particular, that

$$\forall E \in \textit{ENVS},\ F \in \textit{FIELDS},\ P \in \textit{PAIRS} \quad (S4\text{-}230)$$
$$[[\Phi EF(\Psi EF(CDR(P,F)) = \Phi EF(CDR(P,F))] \wedge$$
$$[NORMAL\text{-}FORM(\Psi EF(CDR(P,F)))]]$$

Thus we can assume that

$$NORMAL\text{-}FORM(S_3) \quad (S4\text{-}231)$$

and that

$$\forall E \in \textit{ENVS},\ F \in \textit{FIELDS}\ [\Phi EF(S_3) = D_3] \quad (S4\text{-}232)$$

and similar versions for S_2 and D_2, in the appropriately scoped contexts. In addition, since CAR(X,F) is a partial function defined only over X in ***S***, we can assume that D_2 is in ***S***. In addition, since $\Phi EF(S_3) = D_2$, because of the semantical type theorem we know that S_3 is a handle. (In fact we have not proved the semantical type theorem, but it would be proved in step with the normalisation theorem we are currently proving — hence it is legitimate to assume its truth *on the arguments* to CAR, since we are taking ourselves to be illustrating not a full proof but the proof of one step of an encompassing inductive proof.) Finally, because of the declarative import of handles, we know that:

$$[HANDLE^{-1}(S_3) = D_3] \quad (S4\text{-}233)$$

From all of these it follows that:

$$\forall F \in \textit{FIELDS}\ [CAR(HANDLE^{-1}(S_3),F) = CAR(D_3,F)] \quad (S4\text{-}234)$$

and therefore that

$$\forall E \in \textit{ENVS}, F \in \textit{FIELDS} \quad (S4\text{-}235)$$
$$[[S = HANDLE(CAR(HANDLE^{-1}(S_3),F))] \supset$$
$$[\Phi EF(S) = CAR(D_3,F)]]$$

Finally, we know that:

$$[D_2 = D_3] \quad (S4\text{-}236)$$

simply in virtue of the fact that functions yield the same answers for the same inputs. Putting all of this together we have:

$$\forall E \in \textit{ENVS},\ F \in \textit{FIELDS},\ P \in \textit{PAIRS} \qquad \text{(S4-237)}$$
$$[[CAR(P,F) = E_0(\text{"}CAR)] \supset [\Sigma(P,E,F,[\lambda\langle R_1,D_1,E_1,F_1\rangle\ .\ [[NORMAL\text{-}FORM(R_1)] \wedge [\Phi EF(R_1) = D_1]]]]]$$

Discharging the use of Φ yields:

$$\forall E \in \textit{ENVS},\ F \in \textit{FIELDS},\ P \in \textit{PAIRS} \qquad \text{(S4-238)}$$
$$[[CAR(P,F) = E_0(\text{"}CAR)] \supset [\Sigma(P,E,F,[\lambda\langle R_1,D_1,E_1,F_1\rangle\ .\ [[NORMAL\text{-}FORM(R_1)] \wedge [\Sigma(R_1,E_1,F_1,\ [\lambda\langle R_2,D_2,E_2,F_2\rangle\ .\ [D_1 = D_2]])]]])]]$$

It is only a question of introducing the conjunction into the body of the lambda expression to yield:

$$\forall E \in \textit{ENVS},\ F \in \textit{FIELDS},\ P \in \textit{PAIRS} \qquad \text{(S4-239)}$$
$$[[CAR(P,F) = E_0(\text{"}CAR)] \supset [\Sigma(P,E,F,[\lambda\langle R_1,D_1,E_1,F_1\rangle\ .\ \Sigma(R_1,E_1,F_1,\ [\lambda\langle R_2,D_2,E_2,F_2\rangle\ .\ [[D_1 = D_2] \wedge [NORMAL\text{-}FORM(R_1)]]])]]]$$

But this is exactly

$$STANDARD(E_0(\text{"}CAR)) \qquad \text{(S4-240)}$$

Hence we are done.

We will not prove that any other procedures are standard; the proofs would be similar in structure.

Again in a manner exactly parallel to the numeric functions, combinations of the structural selectors can be combined in the usual way: `(CAR (CDR '(A . (B . C))))` designates the atom `B`, and normalises to the handle `'B`, and so forth. From an informal point of view, it seems that 2-LISP works in much the way in which 1-LISP worked, except that it "puts on a quote mark" just before returning the final answer. Some more examples:

```
(CAR '(A B C))                  ⇒   'A                          (S4-241)
(CDR '(A B C))                  ⇒   '[B C]
(CAR (CDR '(A B C)))            ⇒   <TYPE-ERROR>
(CAR (CAR (CAR '(((((A)))))))  ⇒   '(A)
```

Note, incidentally, that whereas in 1-LISP there was some question about the identity of `CAR` and `CDR` of `NIL`, we have no such troubles in 2-LISP, since there is no `NIL`.

4.b.iii. Typing and Identity

Before examining the other simple predicates over the field, it is useful to examine two special primitives, one having to do with the typing of the semantic domain, and the other with object identity. The first is a procedure, called TYPE, that maps its single argument onto one of ten distinguished atoms, depending on the category of the semantic domain into which that argument falls. Twelve semantic categories were listed in S4-144; three of them, however, were types of functions, requiring intensional access to discriminate, as we will examine in a moment. Examples of the behaviour of TYPE on each of the ten primary extensionally discriminable categories are given in the following list:

```
(TYPE 4)          ⇒   'NUMBER                                    (S4-242)
(TYPE [1 2 3])    ⇒   'SEQUENCE
(TYPE $F)         ⇒   'TRUTH-VALUE
(TYPE +)          ⇒   'FUNCTION
(TYPE '4)         ⇒   'NUMERAL
(TYPE 'HELLO)     ⇒   'ATOM
(TYPE '$F)        ⇒   'BOOLEAN
(TYPE '(+ 2 3))   ⇒   'PAIR
(TYPE '[1 2 3])   ⇒   'RAIL
(TYPE ''4)        ⇒   'HANDLE
```

Like the arithmetic functions, TYPE is extensional; thus, although the argument in the first line of this list is a numeral, the designation of (TYPE 4) is the atom NUMBER since that numeral designates a number. (TYPE 4) *normalises* to the handle 'NUMBER, since that is the normal-form designator of the atom NUMBER. Similarly (TYPE [1 2 3]) normalises to the handle designating the atom SEQUENCE, and (TYPE $F) to the handle designating the atom TRUTH-VALUE.

The expression in the fourth line returns the handle 'FUNCTION, because its argument, the atom +, designates a function. Likewise, the last six lines discriminate among the six structural categories: in each case a handle designating an instance of the category is used as the argument to TYPE. The last case is of particular note: the handle ''4 designates the *handle* '4 (which in turn designates the *numeral* 4, which in turns designates the *number* that is the successor of three).

TYPE need not be called with a normal-form designator of its argument, as the following examples illustrate:

```
(TYPE (+ 7 (/ 3 4)))     ⇒  'NUMBER                                    (S4-243)
(TYPE (NTH 1 [6 '6]))    ⇒  'NUMBER
(TYPE (NTH 2 [6 '6]))    ⇒  'NUMERAL
(TYPE (= 1 1))           ⇒  'TRUTH-VALUE
(TYPE $F)                ⇒  'TRUTH-VALUE
(TYPE '$F                ⇒  'BOOLEAN
(TYPE (TYPE $F))         ⇒  'ATOM
(TYPE '(TYPE $F))        ⇒  'PAIR
(TYPE TYPE)              ⇒  'FUNCTION
```

In order to characterise the primitive semantics of TYPE, we first define a corresponding meta-linguistic function of the same name:

```
TYPE : [D → S]                                                        (S4-244)
   ≡ λD . [if [D ∈ S]
              then if     [D ∈ NUMERALS] then "NUMERAL
                   elseif [D ∈ BOOLEANS] then "BOOLEAN
                   elseif [D ∈ ATOMS] then "ATOM
                   elseif [D ∈ PAIRS] then "PAIR
                   elseif [D ∈ RAILS] then "RAIL
                   elseif [D ∈ HANDLES] then "HANDLE
          elseif [D ∈ ABSTRACTIONS]
              then if     [D ∈ INTEGERS] then "NUMBER
                   elseif [D ∈ SEQUENCES] then "SEQUENCE
                   elseif [D ∈ TRUTH-VALUES] then "TRUTH-VALUE
          elseif [D ∈ FUNCTIONS] then "FUNCTION
```

It is then straightforward to define the significance of TYPE within the language, since it is an extensional procedure:

```
SIMPLE("TYPE,                                                         (S4-245)
       TYPE,
       [λS . HANDLE(if [S ∈ HANDLES]
                       then TYPE(HANDLE⁻¹(S)))
                       elseif [S ∈ NUMERALS] then "NUMBER
                       elseif [S ∈ BOOLEANS] then "TRUTH-VALUE
                       elseif [S ∈ PAIRS] then "FUNCTION
                       elseif [S ∈ RAILS] then "SEQUENCE)])
```

We can see here how the ten types emerge from the six structure types: the handles designate six of them (discharged through TYPE of their referent), and four of the other five categories designate the other four semantical types. Since atoms are not normal-form designators, no check need be made for them explicitly.

From S4-244 and S4-245 all of the behaviour in S4-242 and S4-243 follows directly; we need say no more to characterise TYPE fully.

There is, however, a comment worth making about the use of TYPE over functions. We have sorted *procedures* into three categories: EXPRS, IMPRS, and MACROS; it is natural to

wonder whether a better definition of TYPE might be defined, to return one of the three atoms EXPR, IMPR, and MACRO, rather than the undifferentiating atom FUNCTION. Thus, on this view, it might seem that we would prefer the following behaviour:

```
(TYPE +)          ⇒   'EXPR                                    (S4-246)
(TYPE LAMBDA)     ⇒   'IMPR
(TYPE LET)        ⇒   'MACRO
```

instead of what we have currently defined:

```
(TYPE +)          ⇒   'FUNCTION                                (S4-247)
(TYPE LAMBDA)     ⇒   'FUNCTION
(TYPE LET)        ⇒   'FUNCTION
```

This would seem particularly indicated since we have chosen to have TYPE discriminate among the various kinds of s-expressions, and among the various kinds of abstractions, rather than simply designating the atoms S-EXPRESSION or ABSTRACTION.

This option, however, is not easily open to us; such a TYPE would have to be an *intensional* procedure, since, as an argument in chapter 3 showed us, the difference between EXPRS and IMPRS cannot be decided in virtue of designation alone. None of the other categorisations made by TYPE, however, require intensional access to the arguments; it is therefore far more consistent to leave the definition as it is.

There is another reason for this. In 3-LISP, where the TYPE procedure will retain its present definition, it will be possible for the user to define procedure types other than those provided primitively. Thus in order to accommodate an intensional TYPE that sorted among EXPRS and IMPRS we would have, in the latter dialect, to modify it in a generally extensible fashion so as to discriminate among user procedures, which is less than elegant for a primitive procedure. Furthermore, there is no *need* to have the primitive typing predicate make such a discrimination, since the meta-structural capabilities of 2-LISP allow a user-definable procedure to engender just such behaviour. In particular, we can define a procedure called PROCEDURE-TYPE as follows:

```
(DEFINE PROCEDURE-TYPE                                          (S4-248)
   (LAMBDA EXPR [PROCEDURE]
      (SELECT (CAR PROCEDURE)
         [↑EXPR 'EXPR]
         [↑IMPR 'IMPR]
         [↑MACRO 'MACRO]
         [$T (ERROR "Argument was not a closure")])))
```

This examines the *closure* in the function position of the closure to which a function designator normalises (the EXPR, IMPR, and MACRO selectors in the SELECT statement, in other words, designate closures). For example, the atom + (in the initial environment) will normalise, as we have already mentioned, to the closure

```
(<EXPR> E0 [X Y] (+ X Y))                                    (S4-249)
```

Thus in the reduction of the expression (PROCEDURE-TYPE ↑+) the term PROCEDURE will *designate* the closure just described. (CAR PROCEDURE), therefore, will designate the <EXPR> closure. Similarly, (CAR ↑LAMBDA) would designate <IMPR>, and so forth. Definition S4-248, in other words, would generate the following behaviour:

```
(PROCEDURE-TYPE ↑+)          ⇒    'EXPR                      (S4-260)
(PROCEDURE-TYPE ↑LAMBDA)     ⇒    'IMPR
(PROCEDURE-TYPE ↑LET)        ⇒    'MACRO
```

We will from time to time assume this definition in subsequent examples.

Note in S4-248 that PROCEDURE-TYPE is an extensional function, as is TYPE: the difference is that PROCEDURE-TYPE cannot be reduced with function designators: it must be reduced with function designator designators. It is NAME — the up-arrow — that performs the magic of shifting up one level.

Before leaving the discussion of category membership, we will define ten useful utility predicates for use in later examples:

```
(DEFINE ATOM        (LAMBDA EXPR [X] (= (TYPE X) 'ATOM)))          (S4-251)
(DEFINE RAIL        (LAMBDA EXPR [X] (= (TYPE X) 'RAIL)))
(DEFINE PAIR        (LAMBDA EXPR [X] (= (TYPE X) 'PAIR)))
(DEFINE NUMERAL     (LAMBDA EXPR [X] (= (TYPE X) 'NUMERAL)))
(DEFINE HANDLE      (LAMBDA EXPR [X] (= (TYPE X) 'HANDLE)))
(DEFINE BOOLEAN     (LAMBDA EXPR [X] (= (TYPE X) 'BOOLEAN)))

(DEFINE NUMBER      (LAMBDA EXPR [X] (= (TYPE X) 'NUMBER)))
(DEFINE SEQUENCE    (LAMBDA EXPR [X] (= (TYPE X) 'SEQUENCE)))
(DEFINE TRUTH-VALUE (LAMBDA EXPR [X] (= (TYPE X) 'TRUTH-VALUE)))

(DEFINE FUNCTION    (LAMBDA EXPR [X] (= (TYPE X) 'FUNCTION)))
```

Predications in terms of PROCEDURE-TYPE we will do explicitly.

The procedures TYPE and PROCEDURE-TYPE deal with the *category* identity of their arguments. The primitive function that deals most directly with *individual* identity, in distinction, is called "=" — true just in case its two arguments are the same. 2-LISP's = is defined over individual identity; it is therefore like 1-LISP's EQ, not like 1-LISP's EQUAL (we

discuss type-identity in 2-LISP below). Like TYPE, = is an extensional function. Some examples (we have already seen some of these in section 4.a):

```
(= 1 1)                    ⇒   $T                    (S4-262)
(= 1 (- 99 98))            ⇒   $T
(= 1 2)                    ⇒   $F
(= 1 '1)                   ⇒   $F
(= [$T $F] [$T $F])        ⇒   $T
(= '[$T $F] '[$T $F])      ⇒   $F
(= '''2 '''2)              ⇒   $T
```

It might seem that =, from a semantical point of view, would be rather stragithforward, characterised by the following formula:

$$\text{SIMPLE}(\text{"=}, =, [\lambda\langle S_1, S_2\rangle\ T^{-1}[\Phi E_0 F_0(S_1) = \Phi E_0 F_0(S_2)]]) \qquad \text{(S4-263)}$$

We assume we can use E_0 and F_0 immaterially in this equation, since S_1 and S_2 are guaranteed to be context independent designators.

There is a problem, however: the predicate given in S4-263 as the third argument to SIMPLE is not computable. Suppose, in particular, that we call it EQUI-DESIGNATING, and attempt to construct an algorithmic and syntactic definition (the intent is to define a constructive procedure defined over s-expressions that yields the boolean constants $T or $F depending on whether its two normal-form arguments designate the same entity). We would be led to something like the following:

```
EQUI-DESIGNATING : [[S × S] → BOOLEANS]                    (S4-264)
   ≡ λS1.λS2 .
        if [[TYPE(S1) = TYPE(S2)] ∧
           [if [S1 ∈ [HANDLES ∪ BOOLEANS ∪ NUMERALS]]
                 then [S1 = S2]
                 elseif [S1 ∈ RAILS]
                        then [[LENGTH(S1) = LENGTH(S2)] ∧
                              [∀i 1≤i≤LENGTH(S1)
                                 [EQUI-DESIGNATING(NTH(i,S1,F),
                                                   NTH(i,S2,F)) = "$T]]]
                 elseif [S1 ∈ PAIRS] ... ??? ... ]]
           then "$T
           else "$F
```

The difficulty is that we do not have anything to put in case S_1 and S_2 are pairs (when, in other words, they designate functions).

With this predicate, in other words, we encounter our first troubles with the tractability of our definitions. In our mathematical meta-language, we can use equality predicates with relative impunity, but there are of course many cases — functions being the

paradigmatic example — where the identity of two objects is not a decidable question. We have our semantic domain divided into three main types of object: s-expressions, abstractions, and functions. Equality is decidable (and = is therefore defined) over all s-expressions, and over numbers and truth-values and *some* sequences, and not over functions. The difficulty with sequences has to do with the fact that the identity of a sequence is a function of the identity of the elements: equality (at least for finite sequences) can be decided just in case equality of the corresponding members can be decided. The 2-LISP equality predicate, therefore, is defined over all s-expressions and all abstractions, but not over functions (it will produce an error, as shown below). Over sequences there is no guarantee of its being well-defined. Some examples:

```
(= TYPE +)                      ⇒    <ERROR>                    (S4-256)
(= 'TYPE '+)                    ⇒    $F
(= ['TYPE '+] ['TYPE '+])       ⇒    $T
(= ['+ 'TYPE] ['TYPE '+])       ⇒    $F
(= [TYPE +] [TYPE +])           ⇒    <ERROR>
```

We have a choice in deciding how to characterise the semantics of = in light of these tractability problems. We could say that = designates a truth-value just in case its arguments are of a certain form, or we could say that it designates a truth-value just in case the arguments are the same, but that the procedural consequence is simply partial compared with the declarative import. It is the latter approach we will adopt, because our methodological stance is to reconstruct semantical attribution, and there can be no doubt that terms of the form (= X Y) designate truth just in case X and Y are co-designative, whether or not this can be decided by algorithmic means. Thus we are led to the following characterisation:

```
SIMPLE("=,=,                                                    (S4-256)
       λ<N1,N2> . if [[ ΦE0F0(N1) ∈ FUNCTIONS ] ∨ [ ΦE0F0(N2) ∈ FUNCTIONS ]]
                    then <ERROR>
                    else [T⁻¹(ΦE0F0(N1) = ΦE0F0(N2))]))
```

From the fact that = is not defined over functions it should not be concluded that it is not defined over *normal-form function designators* (closures); on the contrary, since these latter are s-expressions, it follows from what we said above that equality is in fact defined in those cases. However, as discussed in the preceding section, the identity of function designators is much finer grained than of the functions they designate, and therefore equality of function designator cannot be used as a substitute for equality of function. In

such a case the *type*-identity of a function designator becomes relevant (in the sense in which we defined a type-identity for 1-LISP lists in chapter 3), since type-identity is a coarser grained metric than strict identity; nonetheless type identity on function designators is still a finer grained metric than identity of function designated.

In order to illustrate this last point, suppose we define a type equality predicate called TYPE-EQUAL. The idea — similar to the definition of EQUAL in 1-LISP — will be to say that numerals, atoms, booleans, and handles are type-identical only with themselves, and that pairs and rails are type-identical just in case their elements (where the CAR and CDR will in this context be taken as elements of a pair) are recursively type-identical. An appropriate definition is given below. We have naturally extended its domain to include not only s-expressions but also numbers, truth-values, and those sequences over whose elements it is defined (we assume in this and other examples that 1ST is (LAMBDA EXPR [X] (NTH 1 X)) and that REST is (LAMBDA EXPR [X] (TAIL 1 X))):

```
(DEFINE TYPE-EQUAL                                                    (S4-257)
  (LAMBDA EXPR [A B]
    (COND [(NOT (= (TYPE A) (TYPE B)) $F]
          [(= (TYPE A) 'FUNCTION)
           (ERROR "only defined over s-expression & abstractions")]
          [(= A B) $T]
          [(MEMBER (TYPE A) '[NUMERAL ATOM BOOLEAN HANDLE]) $F]
          [(= (TYPE A) 'PAIR)
           (AND (TYPE-EQUAL (CAR A) (CAR B))
                (TYPE-EQUAL (CDR A) (CDR B)))]
          [(MEMBER (TYPE A) '[RAIL SEQUENCE])
           (AND (= (LENGTH A) (LENGTH B))
                (AND . (MAP TYPE-EQUAL A B)))])))
```

The penultimate line requires some explanation. AND is a procedure defined over a sequence of any number of arguments: it designates falsity just in case one or more of those arguments designates falsity, truth if all designate truth, and is undefined otherwise. Thus for example we have

```
(AND $T $T $F)                  ⇒   $F                                (S4-258)
(AND (= '[] '[]))               ⇒   $F
(AND)                           ⇒   $T
(AND . (TAIL 1 [$F $T $T]))     ⇒   $T
```

MAP is a function that takes as its first argument a function, that it applies successively to the elements of as many other arguments as it has, stepping down them. The form as a whole designates the sequence of entities designated by the sequence of applications thus generated. Thus for example

```
(MAP (LAMBDA EXPR [X] (+ X 1))                                    (S4-259)
     [10 20 30])                    ⇒    [11 21 31]

(MAP + [1 2 3] [2 3 4])             ⇒    [3 5 7]
```

Thus the expression (AND . (MAP TYPE-EQUAL A B)) will first generate a sequence of booleans depending on the type-identity of the elements of A and B; the whole expression will be true just in case all the elements are type-identical.

We can first illustrate some straighforward uses of TYPE-EQUAL:

```
(TYPE-EQUAL 1 1)                    ⇒    $T                       (S4-260)
(TYPE-EQUAL 1 2)                    ⇒    $F
(TYPE-EQUAL [$T $F] [$T $F])        ⇒    $T
(TYPE-EQUAL '[$T $F] '[$T $F])      ⇒    $T
(TYPE-EQUAL ''[$T $F] ''[$T $F])    ⇒    $F
(TYPE-EQUAL '(CAR X) '(CAR X))      ⇒    $T
```

The real reason we constructed TYPE-EQUAL, however, was to look at the type-equivalence of function designators. Note first that although simple equality is not defined over functions, it is defined over function designators:

```
(= TYPE TYPE)                       ⇒    <ERROR>                  (S4-261)
(= ↑TYPE ↑TYPE)                     ⇒    $T
(= ↑(LAMBDA EXPR [X] (+ X 1))
   ↑(LAMBDA EXPR [X] (+ X 1)))      ⇒    $F
```

As the last example shows, however, it is too fine-grained to count as equivalent even two function designators that have identical spelling. TYPE-EQUAL overcomes this particular limitation, as the following examples illustrate:

```
(TYPE-EQUAL TYPE TYPE)                    ⇒    <ERROR>             (S4-262)
(TYPE-EQUAL ↑TYPE ↑TYPE)                  ⇒    $T
(TYPE-EQUAL ↑(LAMBDA EXPR [X] (+ X 1))
            ↑(LAMBDA EXPR [X] (+ X 1)))   ⇒    $T
```

However even TYPE-EQUAL counts as *different* function designators that not only provably designate the same function, but that on any reasonable theory of intension ought to be counted as *intensionally indistinguishable* as well:

```
(TYPE-EQUAL ↑(LAMBDA EXPR [Y] (+ Y 1))                            (S4-263)
            ↑(LAMBDA EXPR [X] (+ X 1)))   ⇒    $F
```

Thus, as we have several times mentioned, we will not in these dialects be able to provide either extensional or intensional function identity predicates.

It will have been clear to the reader that there is another computational problem that we have consistently ignored: that of non-terminating programs. For example, if X is a rail such that it is its own first element, then (TYPE-EQUAL X Y) will never terminate. In ruling functions out of the source domain for the procedural version of ▪ we were excluding arguments *for which we have no algorithm at all*; the present case is one in which we have a well-defined algorithm that has the property that it will run forever. In constructing meta-theoretic predicates we have attempted to formulate them in ways that are well-behaved in the case of infinite structures; no attempt, however, will be made to define computable predicates guaranteed to work correctly on circular structures (in the sense defined in chapter 2). This may be an area of legitimate study, but it would not be in the spirit of LISP to focus on such issues, since LISP is fundamentally oriented towards tree-structured objects.

A more adequate semantics might map non-terminating expressions onto ⊥; we have mapped (= + +) onto <ERROR>, which is different. Thus the distinction between non-terminating and semantic ill-formedness is maintained in our account. What we have not done — and what we will not do in this investigation — is to make explicit those conditions under which procedural consequence will engender infinite computations. The contribution of 2-LISP is more in the realm of what programs mean than in whether they terminate.

4.b.iv. Selectors on Rails and Sequences

In S4-62, when we first introduced rails, we assumed that our meta-linguistic functions NTH and LENGTH were defined over abstract sequences as well as over 2-LISP rails. When we introduced tails, however, and recognised the mutability of rails in both element and tail position, we were forced to modify our definitions of these two functions to take explicit field arguments. The resultant equations in S4-95 and S4-96 dropped abstract sequences from the domain of the two functions. As we now turn to the use of these functions within 2-LISP, we will again expand their domains to include both sequences and rails, for completeness and convenience. Furthermore, we will assume the same is true for TAIL, in that (TAIL N SEQ) will designate the sequence consisting of the Nth through last elements of the sequence SEQ. It should be recognised, however, that the identity and mutability conditions on syntactic rails and on mathematical sequences are radically distinct.

Before laying out careful characterisations, some examples of the behaviour we will be characterising will be illustrative:

```
(NTH 3 [1 2 3])                          ⇒   3                          (S4-267)
(NTH 3 '[10 20 30])                      ⇒   '30
(NTH 3 ['10 '20 '30])                    ⇒   '30
(NTH 1 (CDR '(FUN ARG1 ARG2)))           ⇒   'ARG1
(TAIL 0 [])                              ⇒   []
(TAIL 0 '[A CASTLE OF PURE DIAMOND])     ⇒   '[A CASTLE OF PURE DIAMOND]
(TAIL 3 '[A CASTLE OF PURE DIAMOND])     ⇒   '[PURE DIAMOND]
(TAIL 5 '[A CASTLE OF PURE DIAMOND])     ⇒   '[]
(CAR . (TAIL 1 ['(A . B) '(C . D)]))     ⇒   'C
(LET [[X '[NEVER MORE]]]
   (= X (TAIL 0 X)))                     ⇒   $T
(LENGTH [])                              ⇒   0
(LENGTH [1 2 3 [4 5]])                   ⇒   4
(LENGTH '[1 '1])                         ⇒   2
(LET [[X '[QUOTH THE RAVEN]]]
   (NTH (LENGTH X) X))                   ⇒   'RAVEN
```

In order to set out the relevant semantics, it is convenient to define a notion of *vector* to subsume both rails and sequences:

VECTORS ≡ ***[RAILS*** ∪ ***SEQUENCES]*** (S4-268)

We can then introduce new versions of the meta-theoretic functions FIRST, REST, NTH, and LENGTH. First we define FIRSTS and RESTS, which are the set of all possible *rail* versions of the two primitive relationships maintained in a field:

$$\begin{aligned} &\textit{FIRSTS} \equiv [\,\textit{VECTORS} \rightarrow [\,\textit{S} \cup \{\bot\}\,]] \\ &\textit{RESTS} \equiv [\,\textit{VECTORS} \rightarrow [\,\textit{RAILS} \cup \{\bot\}\,]] \end{aligned} \qquad \text{(S4-269)}$$

We can then set out four definitions:

$$\begin{aligned} &\texttt{FIRST} : [[\,\textit{VECTORS} \times \textit{FIELDS}\,] \rightarrow \textit{D}\,] \\ &\quad \equiv \lambda V.\lambda F\ .\ \textit{if}\ [V \in \textit{RAILS}]\ \textit{then}\ F^3(V)\ \textit{else}\ V^1 \end{aligned} \qquad \text{(S4-270)}$$

$$\begin{aligned} &\texttt{REST} : [[\,\textit{VECTORS} \times \textit{FIELDS}\,] \rightarrow [\,\textit{VECTORS} \cup \{\bot\}\,]] \\ &\quad \equiv \lambda V.\lambda F\ .\ \textit{if}\ [V \in \textit{RAILS}]\ \textit{then}\ F^4(V) \\ &\qquad\qquad \textit{else}\ \langle V^2, V^3, \ldots, V^k\rangle\ \textit{where}\ k = \texttt{LENGTH}(V) \end{aligned} \qquad \text{(S4-271)}$$

$$\begin{aligned} &\texttt{NTH} : [[\,\textit{INTEGERS} \times \textit{VECTORS} \times \textit{FIELDS}\,] \rightarrow \textit{D}\,] \\ &\quad \equiv \lambda I.\lambda V.\lambda F\ .\ \textit{if}\ [I = 1]\ \textit{then}\ \texttt{FIRST}(V,F)\ \textit{else}\ \texttt{NTH}(I\text{-}1,\texttt{REST}(V),F) \end{aligned} \qquad \text{(S4-272)}$$

$$\begin{aligned} &\texttt{LENGTH} : [[\,\textit{VECTORS} \times \textit{FIELDS}\,] \rightarrow [\,\textit{INTEGERS} \cup \{\bot\}\,]] \\ &\quad \equiv \lambda V.\lambda F\ .\ \textit{if}\ [\texttt{FIRST}(V,F) = \bot] \\ &\qquad\qquad \textit{then}\ 0 \\ &\qquad\qquad \textit{elseif}\ [\exists N\ [\texttt{NTH}(N,V,F) = \bot] \\ &\qquad\qquad\qquad\qquad \textit{then}\ [1 + \texttt{LENGTH}(\texttt{REST}(V,F))] \\ &\qquad\qquad\qquad\qquad \textit{else}\ \infty \end{aligned} \qquad \text{(S4-273)}$$

We need in addition the following constraint saying that rails have firsts and rests together — demonstrably true of F_0 and provably true of all fields constructable in virtue of the form of the primitive operations:

$$\begin{aligned} &\forall R \in \textit{RAILS},\ F \in \textit{FIELDS} \\ &\quad [[\,\exists S_f \in \textit{S}\ [\texttt{FIRST}(R,F) = S_f\,]] \wedge [\,\exists S_r \in \texttt{RAILS}\ [\texttt{REST}(R,F) = S_r\,]]] \vee \\ &\quad [[\,\texttt{FIRST}(R) = \bot\,] \wedge [\,\texttt{REST}(R) = \bot\,]]] \end{aligned} \qquad \text{(S4-274)}$$

The corresponding constraint on sequences is provable:

$$\begin{aligned} &\forall Q \in \textit{SEQUENCES},\ F \in \textit{FIELDS} \\ &\quad [\,\textit{if}\ Q = \langle\rangle\ \textit{then}\ [[\,\texttt{FIRST}(Q,F) = \bot\,] \wedge [\,\texttt{REST}(Q,F) = \bot\,]] \\ &\qquad\qquad \textit{else}\ [[\,\texttt{FIRST}(Q) = Q_1\,] \wedge [\,\texttt{REST}(Q) = \langle Q_2\ Q_3\ \ldots\ Q_k\rangle\,]] \\ &\qquad\qquad\quad \textit{where}\ Q = \langle Q_1\ Q_2\ Q_3\ \ldots\ Q_k\rangle\,] \end{aligned} \qquad \text{(S4-275)}$$

Finally, we can define a meta-theoretic TAIL function:

$$\begin{aligned} &\texttt{TAIL} : [[\,\textit{INTEGERS} \times \textit{VECTORS} \times \textit{FIELDS}\,] \rightarrow [\,\textit{VECTORS} \cup \{\bot\}\,]] \\ &\quad \equiv \lambda I.\lambda V.\lambda F\ . \\ &\qquad \textit{if}\ [I = 0]\ \textit{then}\ V \\ &\qquad\quad \textit{elseif}\ [\texttt{REST}(R,F) = \bot] \\ &\qquad\qquad\qquad \textit{then}\ \bot \\ &\qquad\qquad\qquad \textit{else}\ [\texttt{TAIL}(I\text{-}1,\texttt{REST}(R),F)] \end{aligned} \qquad \text{(S4-276)}$$

Given these new definitions, the primitive selectors overs rails and sequences can then be defined in the same way in which CAR and CDR were defined. Except for the binding of the field argument, we would like:

```
SIMPLE("NTH,                                                    (S4-277)
       [λ<N,X> . NTH(N,X,F)]
       [λ<N,X> . if [X ∈ RAILS]
                    then NTH(M(N),X,F)
                    else HANDLE(NTH(M(N),HANDLE⁻¹(X),F))])

SIMPLE("TAIL,                                                   (S4-278)
       [λ<N,X> . TAIL(N,X,F)]
       [λ<N,X> . if [X ∈ RAILS]
                    then TAIL(M(N),X,F)
                    else HANDLE(TAIL(M(N),HANDLE⁻¹(X),F))])

SIMPLE("LENGTH,                                                 (S4-279)
       [λX . NTH(N,X,F)]
       [λX . if [X ∈ RAILS]
                then LENGTH(X,F)
                else M⁻¹(LENGTH(HANDLE⁻¹(X),F))])
```

Thus instead we have to posit the following (we give the equation only for NTH; the others are similar). Note as usual how the field, tacit in the language itself, is explicit in the meta-theory.

```
Σ(E₀("NTH))                                                     (S4-280)
    = [λE.λF.λC .
         C("(EXPR E₀ [N V] (NTH N V)),
           [λ<S₁,E₁,F₁> .
               Σ(S₁,E₁,F₁,[λ<S₂,D₂,E₂,F₂> .
                              NTH(NTH(1,D₂,F),NTH(2,D₂,F₂),F₂)])],
           E,
           F)]
```

In addition, we need the internalisations of the three primitive closures; this time we take TAIL as our example:

```
Δ[E₀("TAIL)]                                                    (S4-281)
    = λS₁.λE₁.λF₁.λC .
         Σ(S₁,E₁,F₁,
           [λ<S₂,D₂,E₂,F₂> .
               [if [NTH(2,D₂,F₂) ∈ RAILS]
                   then [C(TAIL(NTH(1,D₂,F₂),NTH(2,S₂,F₂),F₂),
                           E₂,F₂)]
                   else [C(HANDLE(TAIL(NTH(1,D₂,F₂),NTH(2,D₂,F₂),F₂)),
                           E₂,F₂)]]])
```

There is some subtlety in this equation that should be explained, that has to do with the fact that TAIL is defined over both rails and sequences. The problem, in brief, is that even if the arguments designate a sequence, there is a question about the relationship between the syntactic identity of the result that is returned in terms of the identity of the argument. In particular, if the argument in a reduction is a normal-form rail, such as [1 2 3], the form

(TAIL [1 2 3]) will return as its result *the actual first tail of its argument*, rather than some different two-element rail [2 3] that *designates* the tail of the sequence designated by the original argument. NTH and TAIL, in other words, have aspects of their procedural consequence above and beyond that implied by their declarative import. The conditional in S4-281 makes this behaviour clear.

In particular, given an application of the form (TAIL <K> <V>), assuming that TAIL has its standard binding, the arguments to TAIL will be normalised, returning a rail consisting of normal-form designators of the index and the vector. For example, if we had

```
(TAIL (+ 0 1) [3 4 5])                                    (S4-282)
```

then the result of normalising the args would be a rail consisting of the integer 1 and the rail [3 4 5]. This is expected, because this rail will *designate* the sequence consisting of the number one and the sequence of the numbers three, four, and five, which are what the original arguments designated. Thus the predicate in the conditional will be true, and the "then" clause will be relevant. The continuation is thus applied to the result of applying the meta-theoretic function TAIL to three arguments: the first is the *number 1* in our example (because the first argument is NTH(1,$\underline{D_2}$,F_2)); the second is the *rail [3 4 5]* (because the second is NTH(2,$\underline{S_2}$,F_2), not NTH(2,D_2,F_2), which might have been expected); and the third is the current field F_2. Thus the application in terms of the meta-theoretic TAIL function will yield the tail [4 5], which will be returned. Because S_2, not D_2, is used, no function like M^{-1} or HANDLE needs to be used to preserve semantic level.

The consequence is that the primitive TAIL procedure, as we noted, is strictly extensional in the sense we defined that term in 3.c.iii: the *referent* of applications formed in terms of it is a function purely of the *designation* of the arguments. In terms of *result*, however, there is a strict dependence of the result on the form of the argument: the actual tail of the normal-form version of the argument will be returned, not simply a co-designative tail. Thus we would have (these will be better explained after RPLACT has been described in section 4.b.vii, but the intent will be clear):

```
> (SET X [1 (+ 1 1) (+ 1 1 1)])                           (S4-283)
> [1 2 3]
> (SET Y (TAIL 2 X))
> [3]                              ; as expected
> (RPLACT 1 †Y '[4 5])             ; extend Y a little
> [4 5]
> Y
```

```
> [3 4 5]                          ; Y of course has been modified
> X
> [1 2 3 4 5]                      ; X has been modified as well
```

It turns out that this behaviour has its very useful aspects; our present concern is merely to illustrate how it is entailed by semantical equation S4-281. Further examples of the interactions of side-effects and general vectors will be investigated in section 4.b.vii.

In the case where TAIL (or NTH) is applied to a rail, the situation is much simpler, and the identity considerations more straighforward, as the "else" part of the conditional in S4-281 makes evident.

Before leaving the subject of vector selectors, we will define a variety of useful procedures in terms of the three primitives, that we will assume in later examples:

```
(DEFINE 1ST                                                    (S4-284)
   (LAMBDA EXPR [VECTOR] (NTH 1 VECTOR))

(DEFINE LAST                                                   (S4-285)
   (LAMBDA EXPR [VECTOR] (NTH (LENGTH VECTOR) VECTOR))

(DEFINE REST                                                   (S4-286)
   (LAMBDA EXPR [VECTOR] (TAIL 1 VECTOR))

(DEFINE EMPTY                                                  (S4-287)
   (LAMBDA EXPR [VECTOR] (= (LENGTH VECTOR) 0))

(DEFINE FOOT                                                   (S4-288)
   (LAMBDA EXPR [VECTOR] (TAIL (LENGTH VECTOR) VECTOR))
```

FIRST, LAST, REST, and EMPTY are self explanatory. FOOT is a procedure, intended to be used primarily over rails, that returns the particular empty rail at the end of a rail. This is useful since RPLACT of the foot of a rail will extend a rail. FOOT will be illustrated in section 4.b.vii.

4.b.v. The Creation of New Structure

Most of the primitive procedures we have introduced — +, -, *, /, CAR, CDR, TYPE, =, and LENGTH — were what we will call procedurally ***simple***, in that their procedural consequence was simply a consequence of their declarative import coupled with the normalisation mandate. Two others — NTH and TAIL — were slightly more involved, in that their procedural consequence involved a preservation of syntactic identity of arguments above and beyond that mandated by declarative import and normalisation requirements. None of these, however, have involved any side effects or creation of new structure. There are two additional classes of structural primitives to be introduced: one having to do with the creation of (or access to) structure otherwise inaccessible, and one having to do with the modification of the mutable relationships in the field. Members of the first class include PCONS, RCONS, SCONS and PREP; of the second, RPLACA, RPLACD, RPLACN, and RPLACT. We will look at each class in turn.

In 1-LISP, the basic structure creation mechanisms were two: the use of parentheses in the lexical notation, and calls to the CONS function. Because 2-LISP has both rails and pairs, there are two notations that create structure: parentheses and brackets. There are in addition two procedures that generate new structure: PCONS, which creates pairs just as in 1-LISP, and RCONS, which creates rails.

Informally, PCONS is rather like 1-LISP's CONS: it is a function of two arguments that engenders the creation of a new pair whose CAR is the structural field element designated by the first argument, and whose CDR is the structural field element designated by the second. The new pair is designated by the whole application; therefore a handle designating the new pair will be returned as the result. RCONS, on the other hand, takes an arbitrary number of arguments, and designates a new rail whose elements are the referents of its arguments. RCONS is not unlike the 1-LISP LIST. Some examples:

```
(PCONS 'A 'B)                    ⇒   '(A . B)                    (S4-291)
(PCONS '+ '[2 3])                ⇒   '(+ 2 3)
(RCONS 'NOW 'IS 'THE 'TIME)      ⇒   '[NOW IS THE TIME]
(PCONS 'NAME (RCONS 'X 'Y))      ⇒   '(NAME X Y)
(RCONS)                          ⇒   '[]
(PCONS)                          ⇒   <ERROR>
```

The 2-LISP PCONS is mathematically described, as in 1-LISP, in terms of the designation of an otherwise inaccessible pair, with the CAR and CDR relationships of the field modified appropriately. More formally, we have the following account. First the full significance:

$$\Sigma(E_0(\text{"}PCONS)) \qquad \text{(S4-292)}$$
$$= [\lambda E.\lambda F.\lambda C\ .$$
$$C(\text{"}(EXPR\ \underline{E_0}\ [A\ B]\ (PCONS\ A\ B)),$$
$$[\lambda\langle S_1,E_1,F_1\rangle\ .\ \Sigma(S_1,E_1,F_1,[\lambda\langle S_2,D_2,E_2,F_2\rangle\ .\ P])]$$
$$E,$$
$$F)]$$
$$where\ [[P \in PAIRS] \wedge [CAR(P,F_2) = D_2^{\,1}] \wedge [CDR(P,F_2) = D_2^{\,2}]]$$

Of more interest is the internalisation, since it is here where the side-effects are manifested:

$$\Delta[E_0(\text{"}PCONS)] \qquad \text{(S4-293)}$$
$$= \lambda S_1.\lambda E_1.\lambda F_1.\lambda C\ .$$
$$\Sigma(S_1,E_1,F_1,[\lambda\langle S_2,D_2,E_2,F_2\rangle\ .\ C(HANDLE(P),E_2,F_3)])$$
$$where\ [[P \in PAIRS] \wedge$$
$$[F_3 = \langle F_a,F_c,F_2^{\,3},F_2^{\,4},F_2^{\,5}\rangle] \wedge$$
$$[INACCESSIBLE(P,E_2,F_2)] \wedge$$
$$[\forall P' \in PAIRS$$
$$[if\ [P' = P]$$
$$then\ [[F_a(P') = HANDLE^{-1}(NTH(1,S_2,F_2))] \wedge$$
$$[F_d(P') = HANDLE^{-1}(NTH(2,S_2,F_2))]]$$
$$else\ [[F_a(P') = F_2^{\,1}(P')] \wedge [F_d(P') = F_2^{\,2}(P')]]]]]$$

Similarly, we have the following significance for RCONS:

$$\Sigma(E_0(\text{"}RCONS)) \qquad \text{(S4-294)}$$
$$= [\lambda E.\lambda F.\lambda C\ .$$
$$C(\text{"}(EXPR\ \underline{E_0}\ ARGS\ (R\ .\ ARGS)),$$
$$[\lambda\langle S_1,E_1,F_1\rangle\ .\ \Sigma(S_1,E_1,F_1,[\lambda\langle S_2,D_2,E_2,F_2\rangle\ .\ R])]$$
$$E,$$
$$F)]$$
$$where\ [[R \in RAILS] \wedge$$
$$[\forall i\ 1 \le i \le LENGTH(D_2,F_2)\ [NTH(i,R,F_2) = D_2^{\,i}]]]$$

Of note in this characterisation is the fact that in the primitive RCONS closure the list of formal parameters is a single atom ARGS, rather than a rail of atoms. As a consequence ARGS will be bound to the arguments as a whole, rather than to them one by one, thus facilitating the use of an indeterminate number of them. This practice is explained in section 4.c.

The following equation expresses the internalisation of the RCONS closure, manifesting the structure creation:

$$\Delta[E_0(\text{"}RCONS)] \qquad \text{(S4-295)}$$
$$= \lambda S_1.\lambda E_1.\lambda F_1.\lambda C\ .$$
$$\Sigma(S_1,E_1,F_1,[\lambda\langle S_2,D_2,E_2,F_2\rangle\ .\ C(\text{HANDLE}(R_0),E_2,F_3)])$$
$$\textit{where}\ [[F_3 = \langle F_2^{\,1},F_2^{\,2},F_f,F_r,F_2^{\,5}\rangle] \wedge$$
$$[\forall i\ 0\leq i\leq \text{LENGTH}(D_2,F_2)$$
$$[\exists R_i \in \textit{RAILS}\ [\text{INACCESSIBLE}(R_i,E_2,F_2)]]] \wedge$$
$$[\forall R' \in \textit{RAILS}$$
$$[\textit{if}\ [\exists i\ 0\leq i\leq[\text{LENGTH}(D_2,F_2)-1]\ [R' = R_i]]$$
$$\textit{then}\ [[F_f(R') = \text{HANDLE}^{-1}(\text{NTH}((i+1),S_2,F_2))] \wedge$$
$$[F_r(R') = R_{i+1}]]$$
$$\textit{elseif}\ [R' = R_k]$$
$$\textit{then}\ [[F_f(R') = \perp] \wedge [F_r(R') = \perp]]$$
$$\textit{else}\ [[F_f(R') = F_2^{\,3}(R')] \wedge [F_r(R') = F_2^{\,4}(R')]]]]]$$

Though these equations completely characterise the designation and import of these two procedures, there are rather a wide variety of consequences that stem from them, which should be illustrated. First, the crucial fact about these two flavours of CONS — this is why the first syllable of "construct" is part of their name — is that on *each* normalisation a different s-expression is designated. This is the weight borne by the term INACCESSIBLE(R,E,F) in the internalised function. In this sense the "declarative" meaning is dependent on the procedural treatment. This is a different kind of dependence than procedures (like PRINT) that have procedural consequence above and beyond their declarative import (applications in terms of PRINT always designate Truth).

The reason we are concerned about the fact that calls to PCONS and RCONS return new s-expressions is of course in case of subsequent side-effects. A striking difference between 2-LISP and 1-LISP has to do with "empty" enumerators. In 1-LISP the atom NIL served as the null list; being an atom, it could not be extended, as the example in S4-67 made clear. Since in 2-LISP all rails can be extended using RPLACT (to be defined below), it is clear that one cannot in general use a constant as the starting element when building up an enumeration. Consider for example the following 1-LISP program to reverse a list:

```
(DEFINE REVERSE (LAMBDA EXPR (L) (REVERSE* L NIL)))                    (S4-296)

(DEFINE REVERSE*                          ; These are 1-LISP           (S4-297)
   (LAMBDA EXPR (OLD NEW)
      (IF (NULL OLD)
          NEW
          (REVERSE* (CDR OLD) (CONS (CAR OLD) NEW)))))
```

A simple-minded translation into 2-LISP would be the following program defined for rails (EMPTY, FIRST, and REST were defined above; PREP, defined below, returns a new rail whose

first element is the first argument and whose first tail is the second):

```
(DEFINE REVERSE (LAMBDA EXPR [L] (REVERSE* L '[])))                    (S4-298)

(DEFINE REVERSE*                                                       (S4-299)
   (LAMBDA EXPR [OLD NEW]
      (IF (EMPTY OLD)
          NEW
          (REVERSE* (REST OLD) (PREP (FIRST OLD) NEW)))))
```

This definition of REVERSE, however, has a bug, as the following session demonstrates:

```
> (SET X (REVERSE '[EXAMPLE FIRST THE IS THIS]))                       (S4-300)
> '[THIS IS THE FIRST EXAMPLE]
> (SET Y (REVERSE '[DIFFERENT IS SECOND THE]))
> '[THE SECOND IS DIFFERENT]
> (RPLACT 6 X '[WITH A NEW TAIL])
> '[WITH A NEW TAIL]
> X                                          ; X is changed, as expected
> '[THIS IS THE FIRST EXAMPLE WITH A NEW TAIL]
> Y                                          ; Y is changed as well!
> '[THE SECOND IS DIFFERENT WITH A NEW TAIL]
```

The problem is that the very same mutable empty rail designated by '[] in the first line of S4-298 is used as the foot of *every rail returned by REVERSE*, and thus any modification of this empty rail will affect every reversed rail. In fact, not only is every other tail produced by this procedure affected, but the rail within the procedure is changed as well. If the body of REVERSE were printed out following the console session just illustrated, the definition would look as follows:

```
(DEFINE REVERSE                                                        (S4-301)
    (LAMBDA EXPR [L] (REVERSE* L '[WITH A NEW TAIL])))
```

A corrected version of S4-298 is the following:

```
(DEFINE REVERSE (LAMBDA EXPR [L] (REVERSE* L (RCONS))))                (S4-302)
```

The definition of REVERSE* can remain as is. With the new definition we would have:

```
> (SET X (REVERSE '[EXAMPLE FIRST THE IS THIS]))                       (S4-303)
> '[THIS IS THE FIRST EXAMPLE]
> (SET Y (REVERSE '[DIFFERENT IS SECOND THE]))
> '[THE SECOND IS DIFFERENT]
> (RPLACT 6 X '[WITH A NEW TAIL])
> '[WITH A NEW TAIL]
> X                                          ; X is changed, as expected
> '[THIS IS THE FIRST EXAMPLE WITH A NEW TAIL]
> Y                                          ; Y is unchanged, as expected
> '[THE SECOND IS DIFFERENT]
```

This is as it should be. The moral is simple: the rationalisation of side effects to work on empty as well as non-empty rails implies that quoted empty rails should be used with caution; expressions of the form (RCONS) are by and large safer. 2-LISP's '[], in other words, is potentially quite different from 1-LISP's '().

It should not be concluded, of course, that there is a *single* empty rail designated by '[], as the following demonstrates:

```
(= '[] '[])          ⇒   $F                                    (S4-304)
```

Rather, the difference between '[] and (RCONS) brings out the difference between structure construction *by the reader* and structure construction *by the processor.* The point is that the *notation* "'[]" causes a new inaccessible rail to be selected by the reader, but the handle that this notation notates forever designates the same empty rail. On the other hand, the string "(RCONS)" notates a pair, each normalisation of which designates a different empty rail. The difference is exactly the same as that between the construction of pairs implied by parentheses and dots, versus the creation of pairs implied by occurences of the procedure PCONS.

Another different between 1-LISP's CONS and 2-LISP's RCONS and PCONS has to do with the inherent typing of the results from 2-LISP's semantical characterisation. In particular, in 1-LISP we have such well-formed evaluations as:

```
(CONS 1 2)           →   (1 . 2)          ; This is 1-LISP     (S4-305)
(CONS T NIL)         →   (T)
```

On the other hand, in 2-LISP all of the following generate type errors:

```
(PCONS 1 2)          ⇒   <TYPE-ERROR>                          (S4-306)
(RCONS $T $F)        ⇒   <TYPE-ERROR>
```

In both cases the functions in question are extensional functions defined over s-expressions, and should therefore be given s-expression designators as arguments. All four arguments in the two examples in S4-306 designate abstract, rather than structural, entities. Of course the following are well-behaved:

```
(PCONS '1 '2)        ⇒   '(1 . 2)                              (S4-307)
(RCONS '$T '$F)      ⇒   '[$T $F]
```

There is more to be said on this subject, however — for example, the facilities demonstrated do not enable us to construct a rail consisting of the numerals designating the

numbers designated by two variables, since the following yields another type error:

```
(LET [[X 3] [Y 4]] (RCONS X Y))      ⇒   <TYPE-ERROR>          (S4-308)
```

and the following attempted solution fails in intent:

```
(LET [[X 3] [Y 4]] (RCONS 'X 'Y))    ⇒   '[X Y]                (S4-309)
```

What we wanted was the rail [3 4], to be designated by the handle '[3 4]. The correct solution, using the primitive naming facility yet to be formally introduced but so often illustrated, is the following:

```
(LET [[X 3] [Y 4]] (RCONS ↑X ↑Y))    ⇒   '[3 4]                (S4-310)
```

In the 1-LISP derived notion of a list, the CONS procedure serves an often-useful function of prepending an element to a list: returning, in other words, a list whose 1ST is its first argument and whose REST is its second. That this was so followed directly from the way in which lists were implemented in 1-LISP, but we do not have access to this solution in 2-LISP, given our separation of pairs and rails. As hinted in the example given in S4-299 above, we use instead a procedure called PREP (pronounced to rhyme with "step" but short for *prepose or prepend*) which is defined to return a new rail whose first element is the referent of its first argument (which must be an s-expression), and whose second tail is the referent of the second argument (which must be a rail). It is possible to define PREP as follows:

```
(DEFINE PREP                                                  (S4-311)
   (LAMBDA EXPR [FIRST REST]
      (LET [[NEW (RCONS FIRST)]]
         (BLOCK (RPLACT 1 NEW REST)
                NEW))))
```

This definition, however, is ugly: it awkwardly uses a side-effect in order to provide a very simple behaviour. In addition, its functionality so very useful — particularly because of its natural use in recursive functions that build up structure, such as the REVERSE example of S4-299 — that we make it a primitive part of 2-LISP. It has already been tacitly implied that our 2-LISP definition is not strictly minimal; this is even more true in 3-LISP, where the basic reflective powers enable one to define non-primitively a variety of procedures one would expect to find as primitive (including SET and LAMBDA). The present example is thus just one example of a situation in which utility over-rules minimalism as a design aesthetic.

It should be noted that, because of the ability to objectify arguments, RCONS serves as a one-level-deep copying function. Suppose in particular that expression X designates a rail; then the expression (RCONS . X) will normalise to a different rail with the same elements, as illustrated in the following examples (actually this works because of a subtlety regarding the relationship between a designator of a rail of s-expressions and a sequence of designators of s-expressions — see section 4.c.iii):

```
> (SET X '[THIS IS A TEST])                                    (S4-312)
> '[THIS IS A TEST]
> (SET Y (RCONS . X))
> '[THIS IS A TEST]
> (= X Y)
> $F
> (RPLACN 2 X 'WAS)
> 'WAS
> X
> '[THIS WAS A TEST]            ; X was modified,
> Y
> '[THIS IS A TEST]             ; but Y, the copy, was not.
```

The same is of course not true of PCONS: if X designates a pair, such as (A . B), then (PCONS . X) will fail, because PCONS expects an argument designating a sequence of two objects, and X designates a pair, not a sequence. However a simple pair copier — or a generalised copier defined over pairs and rails — could readily be defined.

Again it is useful to define a utility — call XCONS — for constructing redexes (not to be confused with MACLISP's XCONS, which is a variant of CONS that takes its arguments in reverse order). (XCONS <F> $\langle A_1 \rangle$ $\langle A_2 \rangle$ ... $\langle A_k \rangle$), in particular, will designate a new redex whose CAR is <F> and whose CDR is the rail [$\langle A_1 \rangle$ $\langle A_2 \rangle$ ... $\langle A_k \rangle$]:

```
(DEFINE XCONS                                                  (S4-313)
   (LAMBDA EXPR ARGS
      (PCONS (1ST ARGS) (RCONS . (REST ARGS)))))
```

Thus for example we would have:

```
(XCONS '+ '1 '2)          ⇒   '(+ 1 2)                         (S4-314)
(XCONS 'RANDOM)           ⇒   '(RANDOM)
(XCONS '1ST '[2 3 4])     ⇒   '(1ST [2 3 4])
```

XCONS will of course be useful primarily in programs that explicitly construct other programs.

4.b.vi. Vector Generalisations

So far our discussions of structure creation have focused exclusively on procedures that designate s-expressions — pairs and rails — since it is only meaningful to talk of creation and accessibility with regard to elements of the structural field. We did not, therefore, mention the creation of new *sequences*: all sequences are mathematically abstract and are assumed to exist Platonically, independent of the state of the field. However we cannot afford to ignore sequences completely, even from the point of view of structure creation, as this section will demonstrate. In section 4.b.iv we defined the term *vector* to include both rails and sequences, and extended the domains of the selectors functions — NTH, TAIL, and LENGTH — to include vectors of both types. It will turn out that we need to define a constructor function SCONS for sequences, and to extend PREP to work over sequences as well..

In section 4.b.iv we commented that the three selector functions — NTH, LENGTH, and TAIL — were defined over both sequences and rails. Thus we had, for example:

```
(LENGTH [])          ⇒  0                                    (S4-315)
(LENGTH '[])         ⇒  0
(NTH 2 [10 20 30])   ⇒  20
(NTH 2 '[10 20 30])  ⇒  '20
```

In the first and third example, a rail was used in order to mention a sequence; in the second and fourth, a handle was used in order to mention a rail. The relevance of these observations here is this: when discussing rail creation in the previous section, we discussed only those circumstances in which rails were *mentioned* (by using handles, and by using applications formed in terms of RCONS and so forth). What we did not discuss was the issue of the identity of rails in contexts where they are *used*, to designate sequences. It is this question that deserves attention.

Some relevant facts have already been stated. For example, we said that some, but not all, rails were normal-form sequence designators. In particular, those whose elements were in normal-form were themselves considered to be in normal form. We stipulated further — and we noted that this was a more stringent requirement on the processor than mere satisfaction of the normalisation mandate — that all normal-form designators normalised to themselves. In other words, *normal-form rails self-normalise*; they do not

simply normalise to an arbitrary normal-form rail designating the same sequence. In other words, although there can exist in the field any number of distinct rails consisting of the numerals 2, 3, and 4, each of them will normalise to itself, rather than to any other. (This is important in part in underlying the fact that multiple normalisations are harmless; the result of the first is the *exactly* the same as that of any further ones, intensionally as well as extensionally.)

This property of 2-LISP's processor can be noticed using, once again, the ubiquitous operator that designates the name of its argument. First, we observe that normal-form rails normalise to lexically indistinguishable rails, as do handles of rails, but not rails that are not in normal-form. In addition, we can see that ↑<EXP> designates a normal-form designator of the referent of its argument — and furthermore, not just *any* normal-form designator of the referents of its argument, but *that normal-form designator to which its argument would normalise* (this will be reviewed in section 4.d):

```
[2 3 4]                 ⇒   [2 3 4]                 (S4-316)
'[2 3 4]                ⇒   '[2 3 4]                (S4-317)
↑[2 3 4]                ⇒   '[2 3 4]                (S4-318)
[2 (+ 2 1) (+ 2 2)]     ⇒   [2 3 4]                 (S4-319)
'[2 (+ 2 1) (+ 2 2)]    ⇒   '[2 (+ 2 1) (+ 2 2)]    (S4-320)
↑[2 (+ 2 1) (+ 2 2)]    ⇒   '[2 3 4]                (S4-321)
```

What is not apparent from these examples, however, is the identity of the rails on the right hand side of these normalisations, in terms of the identity of those on the left. The answers, however, were all predicted by the equations in section S4-105: namely, that in S4-316, S4-317, S4-318, and S4-320 the rail on the right is the same rail as that on the left, whereas in S4-319 and S4-321 it is obviously different. In other words, all the rails that are type-equivalent lexically (in these six examples) are in fact identical, although that is of course not generally true, as the following illustrates:

```
(= [2 3 4] [2 3 4])     ⇒   $T      (S4-322)
(= '[2 3 4] '[2 3 4])   ⇒   $F      (S4-323)
(= ↑[2 3 4] ↑[2 3 4])   ⇒   $F      (S4-324)
```

The ruling equation (provably true for the 2-LISP processor) is this:

$$\forall S \in \mathit{S}\ [\text{NORMAL-FORM}(S) \equiv [\forall E \in \mathit{ENVS},\ F \in \mathit{FIELDS}\ [\Psi EF(S) = S\ ]]] \qquad \text{(S4-325)}$$

This explains how rail identity is preserved in S4-316, S4-317, and S4-320, and why it is not preserved in S4-319 and S4-321. S4-322 through S4-324 are explained because in each case

the two *lexical rail-notators* (elements of *L-RAIL*) notate a distinct rail. S4-322 designates truth because the equality predicate is applied to the designated sequences, not to the designating rail. It is, however, not as immediately obvious that S4-318 can be accounted for by the same considerations.

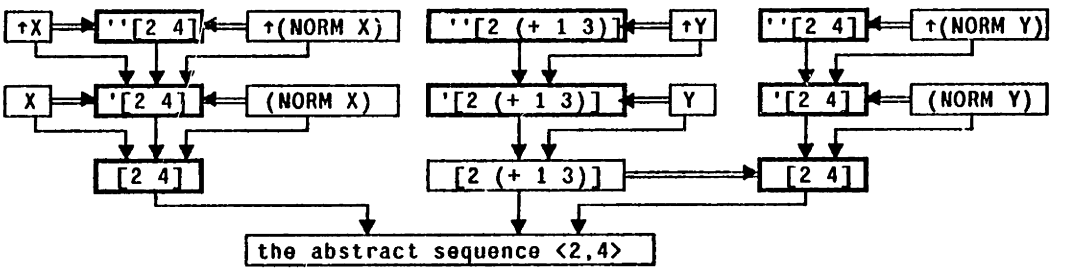
That it does is again predicted by S4-105. Another example illustrating this point is the following:

(S4-326)

```
> (SET X '[2 4])
> '[2 4]
> (SET Y '[2 (+ 1 3)])
> '[2 (+ 1 3)]
> (= X Y)
> $F                                  ; These are of course different rails
> (NORMALISE X)
> '[2 4]                              ; ↑X and ↑Y normalise to equivalent
> (NORMALISE Y)                       ; expressions because the referents of
> '[2 4]                              ; X and Y are co-designative terms.
> (= (NORMALISE X) (NORMALISE Y))     ; However they normalise to
> $F                                  ; different handles.
> (= X (NORMALISE X))                 ; X is self-normalising, because it is
> $T                                  ; in normal form already, whereas
> (= Y (NORMALISE Y))                 ; Y is not self-normalising, because
> $F                                  ; it is not in normal form.
> (LET [[W (NORMALISE X)]]            ; On the other hand both X and Y
      (= W (NORMALISE W)))            ; normalise to self-normalising
> $T                                  ; expressions.
> (LET [[W (NORMALISE Y)]]
      (= W (NORMALISE W)))
> $T
```

The circumstance explored in this dialog is pictured in the following diagram (single-lined arrows signify designation relationships (Φ), double-lined arrows signify normalisation relationships (Ψ), and boxes with heavy outlines are normal-form expressions, which normalise to themselves):

(S4-327)



It is crucial, in interpreting this figure, to recognise that ↑<EXP> designates the result of normalising <EXP>; thus ↑<EXP> normalises to the handle of the result of normalising

<EXP>. Thus ↑[] will always normalise to the same handle. In other words:

```
(LET [[X []]] (= ↑X ↑X))      ⇒      $T                    (S4-328)
```

The consequences of all of this investigation are this: any given empty rail will self-normalise: if the naming operator ↑ is used to obtain mention of this rail, than that rail can be modified. Often this is not a problem, because in a typical procedure body rails are used with variables which are not normal-form; thus if that rail is normalised before being returned, a new rail is generated. Thus we have the following innocuous example (note the use of SET rather than SETQ; this will be explained in section 4.c.vi):

```
> (DEFINE TEST (LAMBDA EXPR [A B] [A B]))                    (S4-329)
> TEST
> (SET X (TEST 3 4))
> [3 4]
> (SET Y (TEST $T $F))
> [$T $F]
> (RPLACT 2 ↑X '[10 20 30])
> '[10 20 30]
> X
> [3 4 10 20 30]
> Y
> [$T $F]                              ; no problems are encountered
> (PRETTY-PRINT TEST)                  ; no problems are encountered

(DEFINE TEST (LAMBDA EXPR [A B] [A B]))
```

The troubles arise only when *empty* sequences are designated. Suppose we for example define a recursive copying procedure intended to construct a new designator of the sequences of its argument's referent in every case, using the PREP function:

```
(DEFINE COPY1                                                (S4-330)
   (LAMBDA EXPR ARGS
      (IF (EMPTY ARGS)
          []
          (PREP (1ST ARGS) (COPY1 . (REST ARGS))))))
```

The intent is to engender the following sorts of behaviour:

```
> (COPY1 1 2 3)                                              (S4-331)
> [1 2 3]
> (SET X ['FOUR 4])
> ['FOUR 4]
> (= X X)
> $T                          ; X of course designates a single thing
> (= ↑X ↑X)
> $T                          ; Furthermore, X is in normal form
> (= X (COPY1 . X))
> $T                          ; They designate the same sequence
> (= ↑X ↑(COPY1 . X))
```

```
> $F                          ; But they are different designators
```

These intended results are all generated by the definition given: $COPY_1$ indeed constructs a new rail, *but only if* ARGS *is non-empty.* Furthermore — against the original intent — every rail returned by $COPY_1$ shares the same foot, as illustrated in:

```
> (= (COPY1) (COPY1))                                          (S4-332)
> $T                          ; This is as it should be
> (= ↑(COPY1) ↑(COPY1))
> $T                          ; But this should be $F
> (SET X (COPY1 3 4))
> [3 4]
> (SET Y (COPY1 5 6))
> [5 6]
> (RPLACT 2 ↑X '[10 20 30])
> [3 4 10 20 30]              ; X has been modified as expected
> Y
> [5 6 10 20 30]              ; But Y has been modified as well
> (PRETTY-PRINT COPY1)        ; So has the definition of COPY1
                              ; (the modified part is underlined)
(DEFINE COPY1
   (LAMBDA EXPR ARGS
      (IF (EMPTY ARGS)
          [10 20 30]
          (PREP (1ST ARGS) (COPY1 . (REST ARGS))))))
```

The solution, instead of using [], is to call the primitive function SCONS, which, like RCONS, returns a *new* (otherwise inaccessible) designator of the sequence of referents of its arguments. For example:

```
> (DEFINE COPY2                                                (S4-333)
     (LAMBDA EXPR ARGS
        (IF (EMPTY ARGS)
            (SCONS)
            (PREP (1ST ARGS) (COPY2 . (REST ARGS))))))
> COPY2
> (SET X (COPY2 3 4))
> [3 4]
> (SET Y (COPY2 5 6))
> [5 6]
> (RPLACT 2 ↑X '[10 20 30])
> [3 4 10 20 30]
> Y
> [5 6]                       ; No problems arise, in Y
> (PRETTY-PRINT COPY2)        ; or in COPY2

(DEFINE COPY2
   (LAMBDA EXPR ARGS
      (IF (EMPTY ARGS)
          (SCONS)
          (PREP (1ST ARGS) (COPY2 . (REST ARGS))))))

> (= (COPY2) (COPY2))
> $T                          ; As expected (both designate the
```

```
> (= ↑(COPY₂) ↑(COPY₂))        ; empty sequence), but they are
> $F                           ; different designators.
```

It should be apparent, in fact, that SCONS *is* COPY_2.

The following examples illustrate in brief how SCONS, RCONS, and PCONS differ:

```
(SCONS)              ⇒   []                                    (S4-334)
(RCONS)              ⇒   '[]
(PCONS)              ⇒   <ERROR: Too few arguments>

(SCONS 'A 'B 'C)     ⇒   ['A 'B 'C]                            (S4-335)
(RCONS 'A 'B 'C)     ⇒   '[A B C]
(PCONS 'A 'B)        ⇒   '(A . B)

(SCONS 1 2 3)        ⇒   [1 2 3]                               (S4-336)
(RCONS 1 2 3)        ⇒   <TYPE-ERROR: Expected an s-expression>
(PCONS 1 2)          ⇒   <TYPE-ERROR: Expected an s-expression>
```

The reader may well wonder whether the distinction between rails and sequences, which apparently gives so much trouble, is worth the effort. The answer is an unqualified *yes*, for a number of reasons. First, we have no choice: it may be that the difference between an abstract sequence of numerals and a rail of numerals is slight, but we simply cannot have a rail of arbitrary objects — like large cities — without violating the very basis of computation. We are forced, in other words, to distinguish structural entities from their referents, by foundational assumptions. That fact, coupled with our inclusion of the structural field in the semantical domain — crucial for such meta-structural considerations in general, and for reflection in particular — leads straight away to the fact that we must encompass both. A possible reply is then that the system might provide automatic conversion between rails and sequences just in case all of the sequence's elements were internal (elements of the field). However this is inelegant and dangerous, in that one is likely to lose any clear sense of the possible range of side-effects.

Furthermore the distinction is far more principled than that between EQ and EQUAL in 1-LISP. In addition, if we include a derivative notion of type-equivalence on vectors, in the way in which we did in order to define 1-LISP's EQUAL — if, in other words, we define a procedure TYPE-EQUAL as in S4-267 — we obtain *three* levels of grain in terms of distinguishing orderings, all semantically well-defined. In fact an infinite number of levels could be distinguished, by defining a type-equivalence predicate defined over vectors whose elements were *individually identical*, and another over vectors whose elements were *type-equivalent* to some degree, and so forth. This suggestion is just the same as the one we

passed by in defining type-equivalence over lists in 1-LISP We again give some illustrations:

```
> (= 1 1)                                                    (S4-337)
> $T
> (TYPE-EQUAL 1 1)
> $T
> (= [1 [] 2] [1 [] 2])
> $F
> (TYPE-EQUAL [1 [] 2] [1 [] 2])
> $T
> (SET X '[THIS IS A RAIL])
> '[THIS IS A RAIL]
> (= X (RCONS . X))
> $F
> (TYPE-EQUAL X (RCONS . X))
> $T
```

We will use these distinctions when appropriate, with due regard for the potential confusions they engender. In practice, such confusions are slight: if one consistently uses (SCONS) in place of [] and (RCONS) in place of '[], all of the identity problems effectively evaporate. Furthermore, the use of TYPE-EQUAL is rarely mandated.

What remain are possible programming use/mention type-errors: using a rail when a sequence was intended, and vice versa. There is no confusion in 2-LISP's behaviour in this regard; quite the contrary: the programmer's problem is usually 2-LISP's unswerving strictness. By and large this will simply be admitted, but one important concession to user convenience will be made, regarding the binding of variables, where a sequence of handles will be made to bind in a manner exactly parallel to a rail of referents. This will be discussed further in section 4.c.iii.

Though RCONS and SCONS are different, we have observed that NTH, TAIL, and LENGTH are defined over vectors of both types. PREP is also defined over both types: it will return a new designator of the same type as its second argument:

```
(PREP 'A '[B C])          ⇒    '[A B C]                    (S4-338)
(PREP 3 [4 5 6])          ⇒    [3 4 5 6]
(PREP 'A (RCONS))         ⇒    '[A]
(PREP $T (SCONS))         ⇒    [$T]
```

Though primitive, even this extended version of PREP could have been (awkwardly) defined as follows:

```
(DEFINE PREP                                                    (S4-339)
   (LAMBDA EXPR [ELEMENT VECTOR]
      (CASE (TYPE VECTOR)
         [RAIL (LET [[NEW (RCONS ELEMENT)]]]
                    (BLOCK (RPLACT 1 NEW VECTOR)
                           NEW))]
         [SEQUENCE (LET [[NEW (SCONS ELEMENT)]]
                        (BLOCK (RPLACT 1 ↑NEW ↑VECTOR)
                               NEW))])))
```

4.b.vii. Structural Field Side Effects

The final category of structural field primitives we have to introduce are those that modify the mutable first-order relationships: CAR, CDR, FIRST, and REST. There are four such functions: RPLACA and RPLACD (as in 1-LISP), that change the CAR and CDR of a pair, and RPLACN and RPLACT, that change elements and tails of rails. Each is defined to "return" the modified pair or rail, having a new CAR, CDR, element, or tail, as the case may be. We have already used all of these functions, in order to demonstrate the salient identity conditions on the structures involved; in this section we will present them more carefully.

RPLACA and RPLACD are as in 1-LISP, except of course they are not used to modify enumerations, since pairs are no longer the basis for an implementation of lists. Some examples:

```
> (SET X '(LENGTH [IT WAS THE BEST OF TIMES]))                    (S4-343)
> '(LENGTH [IT WAS THE BEST OF TIMES])
> (RPLACA X 'TAIL)
> '(TAIL [IT WAS THE BEST OF TIMES])
> (RPLACD X '[2 [CITIES]])
> '(TAIL 2 [CITIES])
```

Because we no longer use lists, and have no distinguished atom NIL, no questions arise about modifying the ends of lists, or of modifying NIL.

Semantically, we expect that the side-effects effected by RPLACA and RPLACD will be manifested in the characterisation of their full procedural consequence, in terms of Σ. In terms of local procedural consequence they are straightforward: they return the normalisation of their second argument. What is rather unclear, however, is what their *declarative* semantics should be, since it is rather unclear that in any natural sense expressions of this sort *stand for* or *refer to* anything: it would seem that their entire import is conveyed in their structural effects.

In 3-LISP we will examine a rather elegant way in which to embody in the formal machine the intuition that expressions with side effects should not be made to designate anything, and similarly not to return anything. When we show how the processor works, we can demonstrate a way in which the primitive procedures that exist in order to change the field can call their continuations with no arguments, implying that their *local* procedural consequence is null (even though their *full* procedural consequence is substantial). We will

at that point define a more sophisticated BLOCK function that demands an "answer" only from the last form within its scope.

For the time being, however, since we don't have machinery for denying a function any declarative import, we will say that each of these (and RPLACN and RPLACT, discussed below) designates the new fragment, thus satisfying the normalisation mandate. In particular, we have the significance/internalisation pair for RPLACA:

$$\Sigma(E_0(\text{"}RPLACA)) \quad \text{(S4-344)}$$
$$= [\lambda E.\lambda F.\lambda C\ .$$
$$\quad C(\text{"}(EXPR\ \underline{E_0}\ [PAIR\ A]\ (RPLACA\ PAIR\ A)),$$
$$\quad [\lambda\langle S_1,E_1,F_1\rangle\ .\ \Sigma(S_1,E_1,F_1,[\lambda\langle S_2,D_2,E_2,F_2\rangle\ .\ D_2{}^2])]$$
$$\quad E,$$
$$\quad F)]$$

$$\Delta[E_0(\text{"}RPLACA)] \quad \text{(S4-345)}$$
$$= \lambda S_1.\lambda E_1.\lambda E_1.\lambda C\ .$$
$$\quad \Sigma(S_1,E_1,F_1,[\lambda\langle S_2,D_2,E_2,F_2\rangle\ .\ C(NTH(2,S_2,F_2),E_2,F_3)])$$
$$\quad where\ [[F_3 = \langle F_a,F_2{}^2,F_2{}^3,F_2{}^4,F_2{}^6\rangle] \wedge$$
$$\quad [\forall P \in PAIRS\ [if\ [P = HANDLE^{-1}(NTH(1,S_2,F_2))]\ .$$
$$\quad then\ [F_a(P) = HANDLE^{-1}(NTH(2,S_2,F_2))]$$
$$\quad else\ [F_a(P) = F_2{}^1(P)]]]]$$

And similarly for RPLACD:

$$\Sigma(E_0(\text{"}RPLACD)) \quad \text{(S4-346)}$$
$$= [\lambda E.\lambda F.\lambda C\ .$$
$$\quad C(\text{"}(EXPR\ \underline{E_0}\ [PAIR\ D]\ (RPLACD\ PAIR\ D)),$$
$$\quad [\lambda\langle S_1,E_1,F_1\rangle\ .\ \Sigma(S_1,E_1,F_1,[\lambda\langle S_2,D_2,E_2,F_2\rangle\ .\ D_2{}^2])]$$
$$\quad E,$$
$$\quad F)]$$

$$\Delta[E_0(\text{"}RPLACD)] \quad \text{(S4-347)}$$
$$= \lambda S_1.\lambda E_1.\lambda F_1.\lambda C\ .$$
$$\quad \Sigma(S_1,E_1,F_1,[\lambda\langle S_2,D_2,E_2,F_2\rangle\ .\ C(NTH(2,S_2,F_2),E_2,F_3)])$$
$$\quad where\ [[F_3 = \langle F_2{}^1,F_d,F_2{}^3,F_2{}^4,F_2{}^6\rangle] \wedge$$
$$\quad [\forall P \in PAIRS\ [if\ [P = HANDLE^{-1}(NTH(1,S_2,F_2))]$$
$$\quad then\ [F_d(P) = HANDLE^{-1}(NTH(2,S_2,F_2))]$$
$$\quad else\ [F_d(P) = F_2{}^2(P)]]]]$$

In both cases the field passed back to the main continuation (as evidenced in the third line of the equation setting forth the internalised function) is not F_2, which is the one received from the normalisation of the arguments, but rather is just like F_2 except that the CAR or CDR of the argument is modified, as one would expect.

The two rail modifiers are more complex, as previous examples have intimated. RPLACN (for "replace NTH") and RPLACT (for "replace tail") each take 3 arguments: the first

designates an index into the rail that is designated by the second argument; the third designates the new element or tail, respectively. The first argument to RPLACN should designate a number between 1 and the length of the rail; the first argument to RPLACT should designate a number between *0* and the length of the rail, since for any rail of length N there are N+1 defined tails. In both cases the modified second argument is returned as the result. We have illustrated both of these functions from time to time in preceding sections; the following examples review their straightforward behaviour:

```
X before:              Form normalised:              X after:          (S4-348)

[CAVE CANEM]           (RPLACN 2 X 'CANTEM)          [CAVE CANTEM]
[]                     (RPLACT 0 X '[NEW TAIL])      [NEW TAIL]
[I LIKE TO WATCH]      (RPLACT (LENGTH X) X '[TOO])  [I LIKE TO WATCH TOO]
[1 2 3 4 5 6]          (RPLACT 3 X (RCONS))          [1 2 3]
```

It is instructive as well to define several standard utility functions on rails, to illustrate the use of these procedures. First we give simple definitions of the 2-LISP rail analogues of 1-LISP's NCONC and APPEND — two procedures that destructively and non-destructively construct the concatenation of two enumerations (we use the term JOIN in place of NCONC; the straightforward COPY is defined in S4-975):

```
(DEFINE JOIN                                                    (S4-349)
   (LAMBDA EXPR [R1 R2] (RPLACT (LENGTH R1) R1 R2)))

(DEFINE APPEND                                                  (S4-350)
   (LAMBDA EXPR [R1 R2] (JOIN (COPY R1) R2)))
```

Equivalently, JOIN can be defined in terms of the FOOT utility defined in S4-288:

```
(DEFINE JOIN                                                    (S4-361)
   (LAMBDA EXPR [R1 R2] (RPLACT 0 (FOOT R1) R2)))
```

Another example is the following procedure, called EXCISE, that takes as arguments an element and a rail and destructively removes any occurences of that element in the rail, splicing the remaining parts together, and returning the number of occurences removed:

```
(DEFINE EXCISE                                                  (S4-362)
   (LAMBDA EXPR [ELEMENT RAIL]
      (EXCISE* 0 ELEMENT RAIL)))

(DEFINE EXCISE*                                                 (S4-363)
   (LAMBDA EXPR [N ELEMENT RAIL]
      (COND [(EMPTY RAIL) N]
            [(= ELEMENT (1ST RAIL))
             (BLOCK (RPLACT 0 RAIL (REST RAIL))
                    (EXCISE* (+ N 1) ELEMENT RAIL))]
```

```
[$T (EXCISE* N ELEMENT (REST RAIL))])))
```

For example:

```
> (SET X '[I'LL NEVER SAY NEVER AGAIN AGAIN])                    (S4-364)
> '[I'LL NEVER SAY NEVER AGAIN AGAIN]
> (EXCISE 'NEVER X)
> 2
> X
> '[I'LL SAY AGAIN AGAIN]
```

Straightforward as these examples seem, there are some subtleties that emerge on a closer look, worth mentioning particularly because they yield behaviour substantially different from that of 1-LISP. In particular, we have pointed out that the ability to use (RPLACT 0 ...) to change *all* of a rail is a facility that 1-LISP did not have; the consequences of this behaviour, however, are visible even if 0 is not used as a RPLACT index. Consider for example the following session:

```
> (SET X '[IF NOT BECAUSE])                                      (S4-366)
> '[IF NOT BECAUSE]
> (SET Y (TAIL 1 X))
> '[NOT BECAUSE]
> (RPLACT 1 X '[AND ONLY IF])
> '[AND ONLY IF]
> X
> '[IF AND ONLY IF]                    ; as expected
```

The question, however, is what Y designates in the resultant context. If this were 1-LISP, and lists were being used in place of rails, the answer would clearly be the "list" (NOT BECAUSE). In other words we have the following:

```
> (SET X '(IF NOT BECAUSE))            ; This is 1-LISP          (S4-366)
> (IF NOT BECAUSE)
> (SET Y (CDR X))
> (NOT BECAUSE)
> (RPLACD X '(AND ONLY IF))
> (AND ONLY IF)
> X
> (IF AND ONLY IF)                     ; as expected
> Y
> (NOT BECAUSE)                        ; Y doesn't see the change to X
```

In some sense we have an option in 2-LISP — in that we could simply *posit* that in the example given in S4-366 Y should designate [NOT BECAUSE] — but this would mean that RPLACT, if its first argument was other than zero, would have a discernably different behaviour from that when its argument is zero. Such a design choice would nullify all of the cleanliness we obtained by making our procedures work equivalently at any rail

position, rather than having to specify particular behaviour at the beginning and end, in the 1-LISP fashion. Therefore the only acceptable choice is to have RPLACT uniform, implying that Y at the end of S4-365 should be bound to the handle '[AND ONLY IF].

Such a choice, moreover, represents no loss of power, for a procedure — we will call it REDIRECT — can always be defined that mimics the 1-LISP style of behaviour. We can in particular define the following:

```
(DEFINE REDIRECT                                                    (S4-367)
   (LAMBDA EXPR [INDEX RAIL NEW-TAIL]
      (IF (< INDEX 1)
          (ERROR "REDIRECT called with too small an index")
          (RPLACT (- INDEX 1) RAIL (PREP (NTH INDEX RAIL) NEW-TAIL)))))
```

Thus we would have:

```
> (SET X '[IF NOT BECAUSE])                                         (S4-368)
> '[IF NOT BECAUSE]
> (SET Y (TAIL 1 X))
> '[NOT BECAUSE]
> (REDIRECT 1 X '[AND ONLY IF])
> '[AND ONLY IF]
> X
> '[IF AND ONLY IF]                    ; as expected
> Y
> '[NOT BECAUSE]                       ; Y did not see the redirection of X
```

What is striking about this definition, however, is that it brings with it all of the problems of 1-LISP's RPLACD on lists: it cannot be used on the first element. Thus we are better off in general with our uniform definition.

Semantically, we have some choices as to how to characterise this behaviour. One option would be to encode, within the meta-language, a constructive algorithm that engenders the proper behaviour. Such an algorithm, of course, must be provided by an implementation (one is given in the appendix). It is far simpler, however, to use a non-constructive specification that merely states the constraints that such an implementation must satisfy. From this point of view the behaviour of RPLACT is easily stated: the rail that is changed should simply be *different* in the resultant context. Thus we can simply specify that in the state that results from the execution of a RPLACT instruction, every occurrence of the old tail will be changed to an occurrence of the new rail. This is encoded in the following two equations. The first is straightforward, because it merely manifests the declarative import, which is virtually identical to that of the other modifiers:

```
Σ(E0("RPLACT))                                                    (S4-359)
    = [λE.λF.λC .
        C("(EXPR E0 [INDEX RAIL NEW-TAIL] (RPLACT INDEX RAIL NEW-TAIL)),
          [λ<S1,E1,F1> . Σ(S1,E1,F1,[λ<S2,D2,E2,F2> . D2³])]
          E,
          F)]
```

The work is done in the following:

```
Δ[E0("RPLACT)]                                                    (S4-360)
    = λS1.λE1.λF1.λC .
        Σ(S1,E1,F1,[λ<S2,D2,E2,F2> . C(NTH(3,S2,F2),E3,F3)])
     where let OLD = TAIL(M(NTH(1,S2,F2)),
                          HANDLE⁻¹(NTH(2,S2,F2)),
                          F2),
               NEW = HANDLE⁻¹(NTH(3,S2,F2))
           in [[[E3 ∈ ENVS] ∧ [F3 ∈ FIELDS]] ∧
               [∀A ∈ ATOMS
                  [if [E2(A) = OLD] then [E3(A) = NEW]
                                    else [E3(A) = E2(A)]]] ∧
               [∀S ∈ S, ∀i 1≤i≤δ
                  [if [F2ⁱ(S) = OLD] then [F3ⁱ(A) = NEW]
                                    else [F3ⁱ(A) = F2ⁱ(A)]]]
```

This works because it constrains every possible access to the old tail, and simply states that such accesses will henceforth point to the new rail. It is crucial that there is no way in which an external (i.e., in virtue of lexical notation) structure can reference a rail directly: all occurrences of lexical brackets construct new ones, as we have seen, and other ways in which previously existent rails can be accessed must be mediated by the environment and the field, both of which we have constrained appropriately.

The equations for RPLACN are simpler:

```
Σ(E0("RPLACN))                                                    (S4-361)
    = [λE.λF.λC .
        C("(EXPR E0 [INDEX RAIL ELEMENT] (RPLACN INDEX RAIL ELEMENT)),
          [λ<S1,E1,F1> . Σ(S1,E1,F1,[λ<S2,D2,E2,F2> . D2³])]
          E,
          F)]

Δ[E0("RPLACN)]                                                    (S4-362)
    = λS1.λE1.λF1.λC .
        Σ(S1,E1,F1,[λ<S2,D2,E2,F2> . C(NTH(3,S2,F2),E2,F3)])
     where [[F3 ∈ FIELDS] ∧
            [F3 = <F2¹,F2²,Ff,F2⁴,F2⁵>] ∧
            [∀R ∈ RAILS
               [if [R = TAIL(M(NTH(1,S2,F2)),
                             HANDLE⁻¹(NTH(2,S2,F2)),
                             F2)]
```

$$\textit{then}\ [F_r(R) = \text{HANDLE}^{-1}(\text{NTH}(3, S_2, F_2))]$$
$$\textit{else}\ [F_r(R) = F_2^3(R)]]]$$

As a kind of postscript, we may observe that the 2-LISP rail behaviour can be implemented using a style of "invisible-pointer" mechanism, in the style of the M.I.T. LISP machine.[1] Some subtlety is required, so that chains of invisible pointers do not get constructed containing cycles, but the code is short and straightforward. One final example, of the sort that inadequate implementations are likely to fail on, is the following:

```
> (SET X '[IF NOT TO TURN AGAIN])                                    (S4-363)
> '[IF NOT TO TURN AGAIN]
> (SET Y (TAIL 2 X))
> '[TO TURN AGAIN]
> (RPLACT 2 X '[AT THE BEGINNING])
> '[AT THE BEGINNING]
> Y
> '[AT THE BEGINNING]
> X
> [IF NOT AT THE BEGINNING]
> (SET Z Y)
> '[AT THE BEGINNING]
> (RPLACT 0 Z (TAIL 2 X))
> '[AT THE BEGINNING]                          ; no change to anything
> (RPLACT 2 X '[NOW WHEN])
> '[NOW WHEN]
> Y
> '[NOW WHEN]
> Z
> '[NOW WHEN]
> X
> '[IF NOT NOW WHEN]
```

4.b.viii. Input/Ouput

We have by now dealt with the first five of the categories listed in table S4-165; the last six remain. The three input/output procedures in 2-LISP are sufficiently similar to those in 1-LISP that they can be dealt with quickly, and the conditional, as well, is straightforward (although semantically rather interesting). In this and the next sub-section we will deal with these two groups. The two naming primitives will then be taken up in section 4.c, along with a general discussion of procedure construction, environments, and binding protocols. Finally, the last two categories — the semantical and processor primitives — will occupy our attention in section 4.d.

The three input/output functions of 2-LISP are adapted from 1-LISP, although in each case the argument *designates* the expression printed or read, rather than designating its

referent. This protocol follows from general semantical requirements, and is also the natural design choice, but since it yields somewhat different behaviour from that of 1-LISP, we will look at examples.

In particular, there is a single procedure called (READ) whose procedural consequence is to read in the lexical notation for a single s-expression from the input stream, and to return a designator of that expression as a result. Thus, declaratively, (READ) can be thought of as designating the structure notated by the "next" lexical expression in the input. A console session illustrates (we underline both input and output that is independent of the reading and printing engendered by the READ-NORMALISE-PRINT loop, maintaining the use of italics for input):

```
> (READ) 24                                                    (S4-364)
> '24
> (READ)
            [TIME INEXORABLY DOES ITS THING]
> '[TIME INEXORABLY DOES ITS THING]
> (TYPE (READ))  $T
> 'BOOLEAN                                        ; Not TRUTH-VALUE!
> (+ (READ) (READ)) 4 5
TYPE-ERROR: +, expecting a number, found the numeral '4
```

The reason that the last example produced an error is that what one "reads" are in fact expressions, not signified entities. Not only is this by and large what is wanted, it is also semantically mandated. Suppose in particular that when the s-expression (READ) was normalised the string "(+ 2 3)" was present in the input stream. The s-expression (+ 2 3) could not be returned as the result, since that s-expression is not in normal form. Furthermore, it would be entirely inappropriate to *normalise* that expression, and to return the numeral 5; what we are discussing is simple reading, not a full READ-and-NORMALISE processor. And even from an intuitive point of view, (READ) designates an expression; since the processor is designation-preserving, the normal-form of (READ) should also designate that expression, normally, which is just what the handle '(+ 2 3) does.

That this behaviour is generally appropriate is shown by the following example, illustrative of the sort of code we will encounter when we construct the 2-LISP meta-circular processor. NORMALISE is an extensional function defined over expressions, whose value is the normal-form to which that expression would normalise. We have, for example:

```
> (NORMALISE (READ)) (+ 2 3)                                   (S4-366)
> '5
```

In other words the argument to NORMALISE designates an expression to be normalised; (READ) in this case designates the *pair* (+ 2 3), which normalises to the *numeral* 5 — which numeral is designated by the handle returned. We add numbers, not expressions; we read and normalise expressions, not numbers.

Lest the last example of S4-364 seem awkward from a programming point of view, in spite of this semantical argument, we can again point ahead to a "level-crossing" operator not yet introduced (the inverse of the NAME operator we have used so often). In general, the expression ↓<EXP> will be shown to designate the referent of the referent of <EXP>. Thus the following is facilitated:

```
> (+ ↓(READ) ↓(READ)) 10 20                                    (S4-366)
> 30
```

With regard to printing, there are two primitives: TERPRI, whose procedural consequence is to print an "end-of-line" on the output stream and to return the boolean $T, and PRINT, which prints out the lexicalisation of the expression designated by its argument. Their use almost entirely resembles that of 1-LISP (in this and subsequent examples, structures that are printed by explicit invocations of PRINT, rather than in the normal course of running the READ-NORMALISE-PRINT loop, will be underlined for pedagogical clarity):

```
> (PRINT 'HELLO) 'HELLO                                        (S4-367)
> $T
> (BLOCK (TERPRI) (TERPRI) (PRINT '[MADELEINES AND TEA?]) (TERPRI))

'[MADELEINES AND TEA?]

> $T
> (PRINT '(RPLACN . [2 '[GOOD BYE] 'BUY])) '(RPLACN 2 '[GOOD BYE] 'BUY)
> $T
```

The last example in this set demonstrates that the standard lexical abbreviations are employed by the printing routine when possible.

Although 2-LISP's READ may look from these examples to be (one meta-level) different from the READ in 1-LISP, whereas PRINT looks remarkably similar, the lurking asymmetry is in fact evidence of the semantics of 1-LISP's evaluation, not of 2-LISP's normalisation. It is straightforward that 2-LISP's READ and PRINT are entirely parallel in designation, as illustrated in the following:

```
> (PRINT (READ))    (HELLO) (HELLO)                    (S4-368)
> $T
> (BLOCK (TERPRI)
         (PRINT 'IN:)
         (LET [[X (READ)]]
            (BLOCK (TERPRI)
                   (PRINT 'OUT:)
                   (PRINT X))))
IN:  [DOUBLE TROUBLE]
OUT: [DOUBLE TROUBLE]
> $T
```

Like the various versions of RPLAC-, the output routines are important solely for their effect. Though it is only arguable that (READ) can be said declaratively to *designate* an expression, it is nonetheless evident that it should *return* an expression, especially in LISP's applicative environment. It is not clear, however, that there is any substance in having TERPRI and PRINT return a normal form. Therefore, when we explore the option of having the structural modifiers return no form, we will also make PRINT and TERPRI return no form (i.e., make Ψ*E("(PRINT))* and Φ*E("(TERPRI))* be ⊥ in all environments).

Since we have not included input/output streams in our general significance function, we cannot formally state the consequence of these primitives. The manner in which such characterisations would be made will, however, be clear from our other examples.

The comments made in chapter 3 about the inadequacy of 1-LISP's input/output functions hold equally true for 2-LISP. A practical system would require more flexible primitives — strings as valid elements of the structural field, and perhaps streams as functional objects. It would have to be decided, of course, whether a string is, like a number, an *external* object, in which case a normal-form string designator would have to be selected, or whether they are structural elements (in which case they would be normally designated by handles). However, since our interests lie elsewhere, we will not give input/output considerations any further attention.

4.b.ix. Control

The only control operator we will introduce in 2-LISP is the primitive conditional IF. Since we will be able in 3-LISP to define more radical control functions — THROW and CATCH, QUIT, and so forth — as straight-forward user functions, and will therefore not need to make them primitive in that dialect, there is little point in introducing them in this preparatory version.

We have used the conditional many times already; in normal use it is just like the corresponding conditional in 1-LISP. Some simple examples:

```
(IF (= 1 1) 'YES 'NO)              ⇒   'YES                    (S4-372)
(IF (= 1 '1) 'YES 'NO)             ⇒   'NO
((IF (EVEN 3) + -) 4 5)            ⇒   -1
((NTH 2 [+ IF LAMBDA]) $T 0 1)     ⇒   0
```

As the last two of these illustrate, conditionals can be combined with the higher-order facilities of 2-LISP in potent, if demanding, ways.

We noted in chapter 2 that the computational conditional is semantically striking: *declaratively* it is an extensional function, whereas *procedurally* it is crucially an IMPR, since it must adjust the order in which it processes its arguments (it always processes just two of them: the ***premise*** and one of the two ***consequents***). It in no way examines those arguments, with respect to their form or intension; it merely holds off un-necessary processing, in order to avoid unnecessary side-effects, errors, and potentially non-terminating computations. For example, each of the following examples would engender different behaviour if IF were an EXPR: the first would print an atom; the second would modify the field; and the third would never return:

```
(IF $T 1 (PRINT 'HELLO))                                    (S4-373)
(IF $F (RPLACA X 'TAIL) X)
(IF (= 1 1) (+ X Y) (LENGTH (JOIN R1 R1)))
```

In spite of this, however, conditionals are *extensional*, in the following strict sense: the referent of an application in terms of IF is a function only of the referents of its arguments, not of their intensional form (although, like all extensional functions, it is a function of their reference in a possibly modified context of use).

Our characterisation of the procedural aspects of the conditional will involve us in some complexities, having to do with unexpected interactions between argument

objectification and non-standard order of processing. These are best illustrated by presenting a natural semantical account and showing how it yields unacceptable behaviour. Given what we have said about how IF processes its arguments, the straightforward semantical characterisation would seem to be the following (we will call this version of the conditional IF_1, since we will look at other proposals presently):

```
Σ[E0("IF1)]                                                          (S4-374)
    = [λE.λF.λC .
        C("(IMPR E0 [PRED A B] (IF1 PRED A B)),
          [λ<S1,E1,F1> .
              Σ(NTH(1,S1,F1),E1,F1,
                [λ<S2,D2,E2,F2> .
                    Σ(NTH([if D2 then 2 elseif ¬D2 then 3],S1,F2),
                      E2,F2,[λ<S3,D3,E3,F3> . D3])])]
          E,F)]

Δ[E0("IF1)]                                                          (S4-375)
    = λS1.λE1.λF1.λC .
        Σ(NTH(1,S1,F1),E,F
          [λ<S2,D2,E2,F2> .
              if     [S2 = "$T] then Σ(NTH(2,S1,F2),E2,F2,C)
              elseif [S2 = "$F] then Σ(NTH(3,S1,F2),E2,F2,C)])
```

IF_1, in other words, is on this account bound in the initial environment to a primitive closure that designates a conditional function. Note, however, that even the declarative designation of the IF_1 closure must explicitly obtain the possibly modified context engendered by processing the first argument (the premise), in order to obtain the designation of the appropriate consequent in that context. The behaviour would be different if, instead, the equation were the simpler:

```
Σ[E0("IF1)]                                                          (S4-376)
    = [λE.λF.λC .
        C("(IMPR E0 [PRED A B] (IF1 PRED A B)),
          [λ<S1,E1,F1> .
              Σ(S1,E1,F1,[λ<S2,D2,E2,F2> . if D2¹ then D2² else D2³])]
          E,F)]
```

since this would make the context in which the second consequent was examined potentially vulnerable to unwanted side-effects of normalising the first consequent. It would imply, for example, that

```
(LET [[X 2]] (IF1 $F (SET X 3) X))                                   (S4-377)
```

would designate the number three, whereas on the account we have given this designates two: and it certainly normalises to 2.

The internalised function given in S4-375 is essentially similar in structure to the full significance: it too tests the first argument, although of course it checks to see whether the result of the premise term is a boolean, whereas the function designated by the closure can be defined in terms of the actual truth value designated by that premise.

A variety of things should be noted about the two equations. First, the semantical typing of 2-LISP requires that the first argument designate one of the two truth-values; thus one does not have the freedom, as one does in 1-LISP, of using any expression other than one designating truth, in first position, in order to have the second expression normalised:

```
> (IF1 (= 2 3) (+ 2 3) (- 2 3))                                    (S4-378)
> -1
> (IF1 (+ 2 3) 'YES 'NO)
TYPE-ERROR: IF1, expecting a truth-value, found the number 5
```

Secondly, LISPs in general, and 2-LISP in particular, process the arguments to EXPR procedures in what is called *applicative* order, as described in chapter 3. The conditional, on the contrary, employs what in the lambda calculus is called *normal* order; it is this ordering difference that enables unwanted processing to be avoided. This fact is reflected in the equation S4-375 in which it is obvious that only one of the two consequences will be processed, depending on the result of normalising the first.

These two points are merely observations: the third begins with an observation, but it is rather serious in consequence. It turns out that there is a rather curious interaction between any IMPR, of which the conditional is a paradigmatic example, and our touted ability to give as the CDR of an application a non-rail expression that nonetheless *designates* an appropriate sequence of arguments. Suppose, for example, that the variable X designated the sequence of Truth, the number 1, and the number 2. We have the following rather disquieting result (in the first example we avoid the standard lexical abbreviation — $(F_1\ F_2\ \ldots\ F_k)$ for $(F_1\ .\ [F_2\ \ldots\ F_k])$ — in order better to exhibit the structure under analysis):

```
> (IF1 . [$T 1 2])                                                 (S4-379)
> 1
> (SET X [$T 1 2])
> [$T 1 2]
> (IF1 . X)
<ERROR: IF1, expecting a rail, found the atom 'X>
```

The problem is that IF_1 cannot select the first of three arguments for normalisation, since X is not a rail. This is predicted by equation S4-374 and S4-375, which as written apply the meta-theoretic NTH function to the total, non-processed, argument, in order to extract the first argument (the premise term) for normalising. In the example shown in S4-379, the argument structure designated by S_1 in for example equation S4-375 will be the atom X, which is of the wrong type for an argument to NTH.

The exact nature of the trouble is most clearly demonstrated by a set of further examples. In the illustrations below we have chosen consequents with obvious side-effects, so that it is immediately apparent when they are processed. (In addition, the expressions printed by such processing are underlined, and we once again avoid the standard lexical abbreviation.) First we duplicate the basic character of our results so far:

```
> (IF1 . [$T (PRINT 'HELLO) (PRINT 'GOOD-BYE)]) HELLO                (S4-380)
> $T
> (SET Y [$T (PRINT 'HELLO) (PRINT 'GOOD-BYE)] HELLO GOOD-BYE
> [$T $T $T]
> (IF1 . Y)
<ERROR: IF1, expecting a rail, found the atom 'Y>
```

The first four lines here are as we would expect: the fifth and sixth are what is troublesome, even though the equations in S4-374 and S4-375 predict it. What makes the situation even more confusing is the fact that by employing meta-structural machinery it seems possible to get around the odd behaviour illustrated in S4-379 and S4-380. We have in particular the following (continuing with the same binding of Y):

```
> ↓(PCONS 'IF1 ↑Y)                                                    (S4-381)
> $T
```

or equivalently (REDUCE will be explained in section 4.d.ii):

```
> ↓(REDUCE 'IF1 ↑Y)                                                   (S4-382)
> $T
```

What is going on here is this: the variable Y is bound to a rail of booleans. Because IF_1 does not first process its arguments, Y cannot be used in the CDR of a procedure reduction. However if *we* construct a procedure ourselves, using PCONS explicitly, and put down as the CDR not the atom Y but the rail to which it is bound, we can of course bypass the problem. In particular, the pair generated by (PCONS 'IF_1 ↑Y) is (IF $T $T $T), which of course is normalised without trouble. The second alternative also bypasses the problem: since ↑Y is processed upon the call to REDUCE, that function is given the conditional and the rail [$T $T

$T] as arguments, which are again handled without trouble.

Of course there is something odd about these solutions, since the processing of the PRINTs happened when Y was bound. But the meta-structural approach also enables us to construct a pair without processing the two printing requests first: we need to bind a variable (we will use Z) to the *handle* designating the appropriate rail, rather than to the rail itself. In particular, we can have:

```
> (SET Z '[$T (PRINT 'HELLO) (PRINT 'GOOD-BYE)])                    (S4-383)
> '[$T (PRINT 'HELLO) (PRINT 'GOOD-BYE)]
> (IF1 . Z)
<ERROR: IF1, expecting a rail, found the atom 'Z>
> ↓(PCONS 'IF1 Z) HELLO
> $T
> ↓(REDUCE 'IF1 Z) HELLO
> $T
```

In other words on this approach we can even manage to have just one of the two expressions processed, depending on the result of normalising the premise.

We have stumbled on what is a remarkably deep problem: our first serious challenge — mentioned in the first chapter — to the claim that objectification can be achieved in an intensional object language (for it is IF_1's procedural intensionality that is causing the problems). A variety of potential solutions present themselves. One would be simply to live with the situation as described: one could argue that it will arise, after all, only in the case of IMPRs, which are presumably less common than EXPRs. Furthermore, it arises only when IMPRs and objectified arguments are *combined*, making trouble an even less likely occurence. Finally, as the examples just cited illustrate, meta-structural machinery apparently enables one to get around the problem in cases where it does arise.

This is a totally unacceptable suggestion, however, for a number of both aesthetic and theoretical reasons. For one thing, it represents an abandoning of effort — a conclusion we should adopt, if ever, only after considerably more investigation. Furthermore, it will be the case in a higher-order dialect like 2-LISP that procedures will be passed as arguments, and it would be nice to be able to use objectified arguments *without knowing the intensional (procedure) type* of the function being called. For example, it is easy to image a definition of MAP along the following lines (this is far from efficient, but it is relatively easy to understand, and it works):

```
(DEFINE MAP                                                    (S4-384)
   (LAMBDA EXPR ARGS
      (MAP* (FIRST ARGS) (REST ARGS)))))

(DEFINE MAP*                                                   (S4-385)
   (LAMBDA EXPR [FUN VECTORS]
      (IF (EMPTY VECTORS)
          (FUN)
          (PREP (FUN . (FIRSTS VECTORS))
                (MAP* FUN (RESTS VECTORS))))))

(DEFINE FIRSTS                                                 (S4-386)
   (LAMBDA EXPR [VECTORS]
      (IF (EMPTY VECTORS)
          VECTORS                          ; Handles both rails and sequences
          (PREP (FIRST (FIRST VECTORS))
                (FIRSTS (REST VECTORS))))))

(DEFINE RESTS                                                  (S4-387)
   (LAMBDA EXPR [VECTORS]
      (IF (EMPTY VECTORS)
          VECTORS
          (PREP (REST (FIRST VECTORS))
                (RESTS (REST VECTORS))))))
```

This definition would support the following:

```
(MAP + [1 2 3] [4 5 6])              ⇒   [5 7 9]                (S4-388)
(LET [[X '[ONCE UPON A TIME]]]
   (MAP NTH [4 3 1] [X X X]))        ⇒   ['TIME 'A 'ONCE]
```

However it would not support

```
(MAP IF1                                                       (S4-389)
     [(= 1 1) (= 1 '1)]
     ['A 'B]
     ['C 'D])                        ⇒   ['A 'D]
```

as we might reasonably expect, given the fact that FUN in S4-386 is called with a non-rail CDR. Nor does it seem reasonable to require that MAP distinguish procedure type: there is nothing to prevent the following, for example:

```
(DEFINE INCREMENT                                              (S4-390)
   (LAMBDA MACRO [X] `(+ ,X 1))

(MAP INCREMENT [1 2 3])              ⇒   [2 3 4]                (S4-391)
```

and MACROs are procedurally distinct from EXPRs. If IMPRs must be especially excluded from such general company, a better reason needs to be offered than we have yet put forward.

Furthermore, our retreat to meta-structural machinery is much less general than the examples so far presented might suggest. The trouble here stems from the static scoping

protocols of 2-LISP that are part of its general ability to deal conveniently with higher-order functions. In particular, we said that

```
↓(PCONS 'IF₁ Y)                                                    (S4-392)
```

where Y was bound to

```
'[$T (PRINT 'HELLO) (PRINT 'GOOD-BYE)]                             (S4-393)
```

would have the same general significance as

```
(IF₁ $T (PRINT 'HELLO) (PRINT 'GOOD-BYE))                          (S4-394)
```

However the fact that this works is due in part to the fact that there are no free variables within the scope of Y. If we use a slightly more complex example, we will not be so lucky. For example, the following is straightforward:

```
(LET [[X 3] [Y 4]]                                                 (S4-395)
   (IF (= X Y) X Y))                ⇒   4
```

However the expression

```
(LET [[X 3] [Y 4]]                                                 (S4-396)
   (LET [[Z '[(= X Y) X Y]]]
      ↓(REDUCE 'IF₁ Y)))
```

would generate an error, because the variable Z is bound to a hyper-intensional *expression*, so that when the REDUCE function is given it, the bindings of X and Y will not be available.

In 3-LISP we will have more powerful meta-structural abilities, so that when processing crosses meta-levels in this way explicit environment designators will be available for explicit use. Thus in 3-LISP one could construct the following:

```
(LET [[X 3] [Y 4]]                                                 (S4-397)
  ((LAMBDA REFLECT [[] ENV CONT]                    ; This is 3-LISP
     (LET [[Z '[(= X Y) X Y]]]
        (REDUCE 'IF₁ Z ENV CONT)))))
```

which would first bind ENV and CONT to the environment and continuation in force at the point of reflection, and would subsequently give them as explicit arguments to REDUCE. Thus S4-397 would normalise to 4. But, as we have maintained all along, full procedural reflection is required in order to make sense of meta-structural manipulations in a higher-order calculus with static scoping. Our present task is merely to make sense of objectified arguments to the conditional. What we have illustrated here is that the apparently straightforward resort to meta-structural facilities is in fact not so straightforward: to make

it general awaits 3-LISP. Hence we have yet another argument for finding a palatable solution within the object language.

A second option would be to define a version of IF with more complex behaviour: it could check explicitly to see whether its non-processed argument was a rail, and if so work as indicated above, but if not, it could normalise it explicitly, on the grounds that if someone used (IF . X) there is no other way in which the value of the premise term constituent of X can be determined. Such a conditional — we call it IF_2 — would have the following meta-circular definition (this is easier to comprehend than the corresponding λ-calculus equations in the meta-language). For perspicuity, we have not included other type checking (such as ensuring that PRED returns a boolean) — in a real implementation this would need to be added.

```
(DEFINE IF₂                                                    (S4-398)
   (LAMBDA IMPR ARGS
      (IF (= (TYPE ARGS) 'RAIL)
          (IF (= (NORMALISE (1ST ARGS)) '$T)
              (NORMALISE (2ND ARGS))
              (NORMALISE (3RD ARGS)))
          (LET [[ARGS (NORMALISE ARGS)]]
              (IF (= (1ST ARGS) '$T)
                  (2ND ARGS)
                  (3RD ARGS))))))
```

Note that the inner conditionals cannot be simple "if-then-else" conditionals of the form

```
(IF (NORMALISE (1ST ARGS))                                     (S4-399)
    (NORMALISE (2ND ARGS))
    (NORMALISE (3RD ARGS)))
```

since in a well-formed conditional redex being explicitly processed by the definition of IF_2 given in S4-398, PRED will *designate a boolean*, not a truth-value (since we are *meta*-circular); explicit equality testing is therefore required (i.e., PRED will equal '$T, not $T). Also, the definition as given is more complex than needed: because NORMALISE (2-LISP's Ψ) is idempotent, it is harmless to normalise an expression more than once. Thus the following is equivalent:

```
(DEFINE IF₂                                                    (S4-400)
   (LAMBDA IMPR ARGS
      (LET [[ARGS (IF (= (TYPE ARGS) 'RAIL) ARGS (NORMALISE ARGS))]]
          (IF (= (NORMALISE (1ST ARGS)) '$T)
              (NORMALISE (2ND ARGS))
              (NORMALISE (3RD ARGS))))))
```

The argument for such a conditional is this: in the normal case it is unlikely that this extended behaviour will engender unwanted side-effects, since the normalisation of the CDR is necessary to decipher any meaning of the conditional application as a whole, and presumably it will not in turn spawn further unwanted normalisations. In the following example, for instance, the side-effects involved in normalising the (PRINT ...) expressions happen when X is set, rather than when the conditional is applied:

```
> (SET X [(= 1 2) (PRINT 'YES) (PRINT 'NO)]) YES NO                (S4-401)
> $T
> (IF₂ . X)
> $T                    ; Since all PRINTs normalise to $T
```

However there is still anomalous behaviour where two normalisations are invoked, when the de-referencing operator (↓) is used. Consider for example the following:

```
> (SET Y '[(= 1 2) (PRINT 'YES) (PRINT 'NO)])                       (S4-402)
> '[(= 1 2) (PRINT 'YES) (PRINT 'NO)])
> (IF₂ . ↓Y) YES NO
> $T
```

The normalisation of the expression ↓Y caused the referent of the handle to which Y is bound to be normalised (why this happens is explained in section 4.d); there is a sense in which one might argue that the normalisation implied in the third line of S4-400 should merely have normalised *once*, not twice. In particular, ARGS was bound in example S4-402 to the handle '↓Y; the normalisation of ARGS produced '[$F $T $T] after printing out both YES and NO. What intuitively was wanted was for the normalisation of Y to produce '[(= 1 2) (PRINT 'YES) (PRINT 'NO)].

That we were unable to provide such behaviour is particularly unfortunate given the fact, mentioned earlier, that explicit construction of the procedural reduction (i.e., explicit use of the meta-structural machinery) satisfies this intuition. Not only do we have

```
> ↓(PCONS 'IF₂ Y) YES                                               (S4-403)
> $T
> ↓(REDUCE 'IF₂ Y) YES
> $T
```

but this does not even require IF₂:

```
> ↓(PCONS 'IF Y) YES                                                (S4-404)
> $T
> ↓(REDUCE 'IF Y) YES
> $T
```

Note, however, that we have solved the MAP problem. We have, in particular:

```
(MAP IF₂                                                    (S4-405)
     [(= 1 1) (= 1 '1)]
     ['A 'B]
     ['C 'D])                    ⇒      ['A 'D]
```

In sum, then, IF_2 solves one of our two problems: it allows non-rail total arguments to IF, but it doesn't provide an easy way in which normal-order processing can be used in that situation. Nor do we yet have a solution to the second problem: meta-structural machinery is still necessary to deal with such a circumstance, but, as we pointed out above, meta-structural solutions can be made acceptable only in a reflective dialect.

What then are we to conclude? The situation we find ourselves in is this: declaratively, the conditional is defined over a *sequence* of three entities, but *procedurally* the designators of these entities are processed in a peculiar manner (just two, in fact, are used in any given case). The objectification mandate suggested that a designator *other than a rail* could be used to designate sequences of entities. The conditional has a problem, in such a case, since it needs access to a normal-form designator of the first element of the sequence. *If* three discriminable element designators are provided, then the conditional can process them individually. If, however, such discriminable designators are *not* provided, we were led to conclude that IF was forced to process the full sequence designator, and then to extract the normal-form designator of the *first* sequence element from the *resultant* designator of the whole sequence, which is guaranteed (by the semantical type theorem) to be a rail.

Once we see the issue in this full light, it is clear that the only way in which partial normalisation can occur is for some party to be able to take a sequence designator and dissect it into structurally distinct designators of the sequence elements. The very 2-LISP machine is able to do this with rails — which happen to be the standard sequence designators. In fact if we could *not* pull apart all rails (not just those in normal form) the problem would have arisen for every function we have seen so far, since EXPRs are all called with sequence designators, from which designators of the individual arguments are to be extracted. However in the EXPR case that sequence designator is the result of a normalisation; hence it is guaranteed to be a rail, so we did not encounter the current difficulty. In the procedurally intensional case, at least as we have so far defined it, we

have no protocol for doing this in the general case. Hence we were led to IF_2.

It is natural to ask whether we *could* define such a structural decomposition process in the general case. But in order to ask this question we need to be clearer on what we mean. From one point of view we *do* have a method of taking any sequence designator and yielding a structurally discriminable designator of an element: we normalise the designator as a whole, which is guaranteed to return a rail, which we can then disect. Thus we are led to put our question more carefully as follows: is there any general algorithm by which we can take a sequence designator and, *without processing it*, extract a designator of each argument. But again this needs clarification: what is it to *process* an expression? What we are concerned with, it becomes clear, is that we wish to *avoid any unwanted side-effects* of processing the inappropriate element designator.

It might seem that one way — albeit impractical — would be to suspend the current state of the computation, and to process the whole sequence designator in a completely new and isolated context (into which, say, the whole original context was copied so that the ground will have been appropriately set up). We could then process *all* of the sequence designator, look at the designator of the first element to determine whether the premise came out true or false. It might seem, that is, that if we *knew* whether the premise was true or false "ahead of time", so to speak, we could select the appropriate part of the sequence designator.

But there are two problems. The first has to do with termination: it is not clear that, if any sense could be made of this suggestion, it would be possible to guarantee that this pre-processing "hypothetical" pass could be executed in fashion which would be guaranteed to terminate. Secondly, it simply is not in general definable which parts of the processing of a sequence designator "belong" to which element of the designated sequence. Consider for example the following expression:

```
(BLOCK (SET X 3)                                        (S4-406)
       (SET Y 4)
       (SET Z 6)
       [X Y Z])
```

Any presumption that the expression `(SET X 3)` has to do only with the first element of the resultant sequence is of course based on purely informal and ultimately indefensible assumptions.

There is, in sum, as we said earlier in another context, no solace to be found in this suggestion (perhaps that is fortunate, given the potential complexity of computation it hints at). It should be admitted that for a certain class of sequence designators, however, some approximations to such an approach can be defined. These arise from the notions of "lazy evaluation" and message-based systems.[2] For example, the primitive procedure PREP takes two arguments: one designates the first element of a sequence, one the remaining elements. If all of the elements of a sequence were designated by first arguments in PREP applications, it would be possible to extract those designators one by one. For example, suppose we consider the expression:

```
(IF . (PREP (BLOCK (PRINT 'HI) $T)                                    (S4-407)
            (PREP (BLOCK (PRINT 'THERE) 1)
                  (PREP (BLOCK (PRINT 'HANDSOME) 2)
                        (SCONS)))))
```

Under our original definition of IF, this would generate an error, since the CDR is not a vector. Under IF_2 it would return 1, but only after printing out all three words. Our current suggestion is that it seems just vaguely possible, given that we know how PREP works, that we might arrange it to print out only HI and THERE, given IF's regimen for argument processing.

It is presumably clear that simply positing this one extra condition — that procedural reductions formed in terms of PREP be treated specially — would be inelegant in the extreme; we will not consider it seriously. However it does suggest a tack, far from the LISP style of operation, that we will not adopt, but that might with further investigation lead to a coherent calculus. The suggestion would be to require of all sequence-designating terms that they be able, *in general*, to yield normal-form designators for arbitrary elements (or, to take a more restricted case, for their first element, although this would not solve IF's problems: we would need to have them handle the first *three*). Thus rails would do this by normalising only that element (they are easy); PREP would do it by normalising only its first argument, if the first element was asked; otherwise it would pass the buck to its second argument, with the index reduced by one appropriately, and so forth.

The question, of course, is what *general* procedures would do with such a request (it is easiest to think of this architecture under a message-passing metaphor). Many, such as the primitives for addition and so forth, could legitimately complain (cause an error), since there is no reason they should be asked such a question. User-defined procedures would

pass the query to their bodies, so the question would need to be decided only on the primitives. A crucial question is what BLOCK should do. It is not that the answer is open — there is indeed only one possible answer — rather, the question is whether it makes sense. BLOCK would clearly have to process *in standard fashion* all but the last expression within its scope, and then ask the appropriate question of the final term. Thus in the example shown above in S4-406, all three SETs would happen, before the rail [X Y Z] would be asked for a normal-form designator of its first (or NTH) argument.

It is illuminating to examine the consequences of this suggestion on the examples we have raised, and also to look at another example that is isomorphic, which we used as if it were well-defined in S4-257 above, as the proverbial "astute reader" will have noticed. That example had to do with AND: in section 4.b.iii we rather blithely assumed that a term of approximately the following structure

```
(AND . (MAP EVEN [1 2 3 4 5]))                                    (S4-408)
```

was well-defined. However under normal assumptions (i.e., in traditional LISPs and in 1-LISP) conjunction was defined to process only as many arguments as were necessary until a false one was encountered. Under the obvious definition of AND, given below, we would encounter the problem we have been fighting with the conditional: S4-408 would generate an error because (MAP EVEN [1 2 3 4 5]) is an ill-formed argument for NTH. (Once again we ignore type-checking for perspicuity.)

```
(DEFINE AND1                                                       (S4-409)
   (LAMBDA IMPR ARGS (AND* ARGS)))

(DEFINE AND*                                                       (S4-410)
   (LAMBDA EXPR ARGS
      (COND [(EMPTY ARGS) $T]
            [(= '$F (NORMALISE (FIRST ARGS))) $F]
            [$T (AND* (REST ARGS))])))
```

This definition would support the following:

```
> (AND1 (= 1 1)                                                    (S4-411)
        (BLOCK (PRINT 'HOWDY) $F)
        (BLOCK (PRINT 'STRANGER) $T)) HOWDY
> $F
```

but it would fail on S4-408. We could define an alternative AND_2, by analogy with IF_2, that checked to see whether the arguments were a rail, and if not pre-normalised them, as follows (AND* can remain unchanged):

```
(DEFINE AND2                                                    (S4-412)
   (LAMBDA IMPR ARGS
      (AND* (IF (= (TYPE ARGS) 'RAIL)
                ARGS
                (NORMALISE ARGS)))))
```

This would give us

```
(AND2 . (MAP EVEN [1 2 3 4 5]))      ⇒    $F                    (S4-413)
```

while preserving

```
> (AND2 (= 1 1)                                                 (S4-414)
     (BLOCK (PRINT 'HOWDY) $F)
     (BLOCK (PRINT 'STRANGER) $T)) HOWDY
> $F
```

but it would also generate:

```
> (AND2 . ↓(TAIL 1 '[(BLOCK (PRINT 'NO) $F)                     (S4-415)
     (= 1 1)
     (BLOCK (PRINT 'HOWDY) $F)
     (BLOCK (PRINT 'STRANGER) $T)])) HOWDY STRANGER
> $F
```

With these general examples of the problem set forth, we can return to the suggestion that PREP reductions be dissectable. The problems with this proposal, however, are rather far-reaching. Suppose, for example, that <X> was an expression that we believed designated a sequence, and from which we wanted to extract a designator for the first element. Suppose in addition that this pre-processing of <X> effected a variety of side effects on the resultant context. We may presume that this altered context would be passed back with the designator of the first element. However another structure would have to be created as well, if the remaining elements of the sequence were ever to be determined. For example, if <X> were

```
(BLOCK (SET X (+ X 1))                                          (S4-416)
       (SET Y (+ Y 1))
       (SET Z (+ Z 1))
   (PREP X (PREP Y (PREP Z (SCONS)))))
```

then it would be unacceptable, if a normal-form designator of the second element of the resultant sequence were ever sought, for the three incrementations to be repeated. This is true even if acceptable closure mechanisms and so forth could be introduced to keep the contexts straight. For what this approach is driving towards is a company of generators, with conversations back and forth about pieces of their respective domains. The static

scoping protocols of 2-LISP facilitate this kind of programming, as has often been noted,[3] but to make clear sense of the partial normalisation of sequence designators would require substantial extension of the governing protocols over this underlying behaviour.

Such extensions are not properly part of our current investigation, so we will pursue them no further. What we are left with is a partial solution: we will adopt IF_2 as the primitive 2-LISP conditional, since it deals with half of the troubles with IF_1 and, being a pure extension of that conditional, it does not alter any behaviour obtainable with the simpler version.

There is a question as to whether we should adopt AND_2 as well, in place of the simpler AND_1. On first blush, this would seem consistent; on second blush we realise that AND is not a 2-LISP primitive, and therefore we don't have to decide: we can let the user choose whichever he or she prefers. But yet further consideration should make it evident that AND cannot be adequately defined as an IMPR in 2-LISP at all, for the reason we keep mentioning regarding the use of free variables. In particular, suppose we adopt either definition of AND given in S4-409 or S4-412. Even as simple example as the following will generate an error, because X will be unbound:

```
(LET [[X 3]]                                                    (S4-417)
   (AND $T (= X 2)))
```

The formal parameter ARGS in the definition of AND will be bound to the rail [$T (+ X 2)], which, when given to NORMALISE, requires for its successful treatment the environment that was in effect at the point of normalisation of the call to AND (as do all IMPRS — section section 4.d.iii). Thus an adequate intensional AND awaits 3-LISP also. If we are to have an AND in 2-LISP — of either AND_1 or AND_2 variety — it would seem that it too will have to be primitive.

Fortunately, this is not the case: we can define AND as a macro. The following code, in particular, the details of which will be explained in section 4.d.v, will define a procedurally-intensional conjunction in terms of IF. In other words, given the procedural intensionality of the primitive IF, we can "piggy-back" similar abilities off it with the use of macros. This definition is of the AND_2 variety:

```
(DEFINE AND3                                                    (S4-418)
   (LAMBDA MACRO ARGS
      (IF (= (TYPE ARGS) 'RAIL)
          (AND3^ ARGS)
```

```
          `↓(AND3 ↑,ARGS))))
```

```
(DEFINE AND3*                                                    (S4-419)
   (LAMBDA EXPR [ARGS]
      (IF (EMPTY ARGS) '$T
          `(IF ,(1ST ARGS)
               ,(AND3* (REST ARGS))
               $F))))
```

For example, the following three expressions:

```
(AND3 A B C)                                                     (S4-420)
(AND3 . (MAP EVEN [1 2 3 4 5]))
(AND3)
```

would expand into the following expressions:

```
(IF A (IF B (IF C $T $F) $F) $F)                                 (S4-421)
↓(AND3* ↑(MAP EVEN [1 2 3 4 5]))
$T
```

The first and last of these are staightforward; the second would first normalise to (we make use of the fact that normal-form reductions can simply be substituted into an expression in order to demonstrate a partially processed term, as in the λ-calculus):

```
↓(AND3 '[$F $T $F $T $F])                                        (S4-422)
```

which in turn would normalise to

```
↓'(IF $F (IF $T (IF $F (IF $T (IF $F $T) $F) $F) $F) $F)         (S4-423)
```

which, though ungainly, is semantically justified, and would utlimately yield the proper $F.

The use of macros will be explained more fully in section 4.d.v. It is striking, however, to note in the present context that once we have 3-LISP's primitive reflective abilities, we will need no *primitive* reflective functions at all. IF, AND, and even LAMBDA will be straightforward user-definable reflective procedures (IF can be defined with respect to a primitive but non-procedurally intensional conditional). Once again our analysis has shown that 2-LISP is not on its own a calculus with natural boundaries.

One task remains: to demonstrate the proper meta-theoretic characterisation of our revised (IF_2 style) conditional, which we now give:

```
Σ[E0("IF1)]                                                      (S4-424)
   = [λE.λF.λC .
       C("(IMPR E0 [PRED A B] (IF1 PRED A B)),
         [λ<S1,E1,F1> .
            if [S1 ∈ RAILS]
```

```
                then Σ(NTH(1,S1,F1),E1,F1,
                       [λ<S2,D2,E2,F2> .
                           Σ(NTH([if D2 then 2 elseif ¬D2 then 3],S1,F2),
                             E2,F2,[λ<S3,D3,E3,F3> . D3])])
                else Σ(S1,E1,F1,[λ<S2,D2,E2,F2>.if D2^1 thon D2^2 else D2^3])]
            E,F)]

Δ[E0("IF1)]                                                          (S4-426)
   = λS1.λE1.λF1.λC .
       if [S1 ∈ RAILS]
         then Σ(NTH(1,S1,F1),E,F
                [λ<S2,D2,E2,F2> .
                    if     [S2="$T] then Σ(NTH(2,S1,F2),E2,F2,C)
                    elseif [S2="$F] then Σ(NTH(3,S1,F2),E2,F2,C)])
         else Σ(S1,E1,F1,
                [λ<S2,D2,E2,F2> .
                   if     [NTH(1,S2,F2)="$T] then C(NTH(2,S2,F2),E2,F2)
                   elseif [NTH(1,S2,F2)="$T] then C(NTH(3,S2,F2),E2,F2)])
```

A final comment, in passing. This example of interactions between de-referencing, normalisation, and IMPRS illustrates rather vividly the importance of working through a full language design under a set of design mandates. It is straightforward to argue for a design principle like the normalisation mandate and full category alignment, but the ramifications of such a position are rarely evident on the surface. For example, the current difficulty has arisen over the *interaction* between two decisions — one about argument objectification, and one about normalisation — that until this point had seemed compatibile in execution and spirit. The moral is this: although working out the fine grained details of a dialect of LISP might seem a distraction from more important theoretical purposes, particularly the kind of aesthetic detail on which we are spending such time, this example — and many more like it that we will encounter before we have delivered a satisfactory 3-LISP — are in fact crucial to the overall enterprise. All this by way of encouragement to any reader who suspects that we have been seduced into unnecessary technicalities.

4.c. Methods of Composition and Abstraction

We said in chapter 2 that a LISP system is best understood as comprising three types of facilities: a set of primitive objects, and methods of composition and abstraction over these primitives. The first of these aspects of 2-LISP has been presented: in section 4.a we introduced the primitive structural types, and in 4.b we defined those primitive procedures that are defined over those structural types. In the present section we turn to the second and third kinds of capability: facilities for the construction of composite entities, and protocols enabling composite objects to be treated as unities. Under this general topic will fall discussions of LAMBDA and closures, naming and variable use, the defining of procedures, the use of environments and global variables, a discussion of recursion, and so forth.

The discussion in this section will focus on object level matters — on issues, in other words, that do not involve meta-structural machinery. Thus, although we will discuss LAMBDA terms, we will by and large restrict our attention to the creation of EXPRS; although we will examine code that *uses* closures, we will not consider programs that *mention* them. The general subject of meta-structural facilities of 2-LISP will be considered separately, in section 4.d.

4.c.i. Lambda Abstraction and Procedural Intension

Atoms, as we said in section 4.a, are used in 2-LISP as context dependent names. We also made clear, both in that section and in the discussion in chapter 3, that they are taken to designate the referent of the expression to which they were bound. Finally, we have said that they will be statically scoped. It is appropriate to look at all of these issues with a little more care.

The semantical equation governing atoms was given in section 4.a.iii; we repeat it here:

$$\forall E \in \textit{ENVS},\ F \in \textit{FIELDS},\ C \in \textit{CONTS},\ A \in \textit{ATOMS} \qquad \text{(S4-430)}$$
$$[\,\Sigma(A,E,F,C) = C(E(A),\Phi EF(E(A)),E,F)\,]$$

If we discharge the use of the abbreviatory Φ, this becomes:

$$\forall E \in \textit{ENVS},\ F \in \textit{FIELDS},\ C \in \textit{CONTS},\ A \in \textit{ATOMS} \qquad \text{(S4-431)}$$
$$[\,\Sigma(A,E,F,C) = C(E(A),\ \Sigma(E(A),E,F,[\lambda\langle S,D,E_1,F_1\rangle\ .\ D]),$$

E,F)]

Because all *bindings* are in normal-form, the above equation can be *proved* equivalent to the following:

∀E ∈ ***ENVS***, F ∈ ***FIELDS***, C ∈ ***CONTS***, A ∈ ***ATOMS*** (S4-432)
[Σ(A,E,F,C) = Σ(E(A),E,F,C)]

This is true because, if E(A) is normal, then it will not affect the E and F that are passed to it. Nonetheless, S4-431 must stand as the definition; S4-432 as a consequence.

What we did not explain, however, is how environments are constructed. The answer, of course, has first and foremost to do with λ-binding. A full account of the significance of atoms and variables, therefore, must rest on the account of the significance of λ-terms. A λ-term is, in brief, an expression that designates a function. Structurally, it is any reduction (pair) formed in terms of a designator of the primitive lambda closure and three arguments: a *procedure type*, a *parameter list*, and a *body expression*. The primitive lambda closure is the binding, in the initial environment, of the atom LAMBDA, although there is nothing inviolate about this association. The procedure type argument is typically either EXPR or FEXPR — we will discuss what these terms mean below. The parameter list is a pattern against which arguments are matched, and the body expression is an expression that, typically, contains occurences of the variables named in the parameter pattern. Thus we are assuming λ-terms of the following form:

(LAMBDA <PROCEDURE-TYPE> <PARAMETERS> <BODY>) (S4-433)

We have of course used λ-terms throughout the dissertation, both in LISP and in our meta-language. We must not, however, be misled by this familiarity into thinking we either understand or have encountered the full set of issues having to do with LAMBDA abstraction. For this reason we will assume in the following discussion that LAMBDA is being for the first time introduced. In this spirit, we do well to start with some examples of the use merely as embedded terms (i.e., without any of the complexities of global variables, top-level definitions, recursion, or the like). These examples are similar in structure to the kind of term that can be expressed in the λ-calculus:

((LAMBDA EXPR [X] (+ X 1) 3) ⇒ 4 (S4-434)

```
((LAMBDA EXPR [F]                                                    (S4-436)
   (F (F 3 4) (F 5 6)))
 +)                                    ⇒    18

((LAMBDA EXPR [G1 G2]                                                (S4-436)
   (G1 (= (NTH 1 '[$T])
          (NTH 1 ['$T]))
       (G2 [10 20 30])
       (G2 '[10 20 30])))
 IF
 (LAMBDA EXPR [R] (TAIL 2 R)))         ⇒    [30]
```

S4-434 is a standard example, of the sort 1-LISP would support: the expression (LAMBDA EXPR [X] (+ X 1)) designates the increment function. S4-435 illustrates the use of a function designator as an argument, making evident the fact that 2-LISP is higher order. Finally, S4-436 shows that procedurally intensional designators (IF) can be passed as arguments as readily as EXPRs.

There is nothing distinguished or special about these LAMBDA expressions, other than the fact that LAMBDA designates a primitive closure Unlike standard LISPs and the original λ-calculus, in other words, the label LAMBDA is not treated as a syntactic mark to distinguish one kind of expression from general function applications. LAMBDA terms in 2-LISP are reductions, like all pairs, in which the procedure to which LAMBDA is bound is reduced with a standard set of arguments. We will see below that LAMBDA is initially bound to an intensional procedure, but, as example S4-437 demonstrates, this fact does not prevent that closure from itself being passed as an argument, or bound to a different atom:

```
(((LAMBDA EXPR [F]                                                   (S4-437)
    (F EXPR [Y] (* Y Y)))
  LAMBDA)
 5)                                    ⇒    10
```

It happens that EXPR also names a function; thus we can even have such expressions as:

```
(((LAMBDA EXPR [FUNS]                                                (S4-438)
    ((NTH 2 FUNS) (NTH 1 FUNS) [Y] (* Y Y)))
  [EXPR LAMBDA])
 5)                                    ⇒    10
```

Finally, as usual it is the normal-form closures, rather than their names in the standard environment, that are primitively recognised:

```
> (DEFINE BETA LAMBDA)                                               (S4-439)
> BETA
> (DEFINE STANDARD EXPR)
> EXPR
```

```
> ((BETA STANDARD [F] (F F)) TYPE)
> 'FUNCTION
```

LAMBDA, in other words, is a ***functional***: a function whose range is the set of functions:

```
(TYPE LAMBDA)                          ⇒   'FUNCTION          (S4-440)
(TYPE (LAMBDA EXPR [X] (+ X 1)))       ⇒   'FUNCTION
```

Similarly, EXPR is a function, although we will show how it can be used in function position only later:

```
(TYPE EXPR)                            ⇒   'FUNCTION          (S4-441)
```

Though the examples just given illustrate only a fraction of the behaviour of LAMBDA that we will ultimately need to characterise, some of the most important features are clear. First, LAMBDA is first and foremost a *naming* operator: the *procedural* import of LAMBDA terms in this or any other LISP arises not from LAMBDA, but from general principles that permeate structures of all sort, and from the type argument we have here made explicit as LAMBDA's first argument. We will explore the procedural significance of LAMBDA terms at length, but it is important to enter into that discussion fully recognising that it is the body expression that establishes that procedural import, not LAMBDA itself.

Second, LAMBDA is itself an *intensional* procedure; neither the parameter pattern nor the body expression is processed when the LAMBDA reduction it itself processed. This is clear in all of the foregoing examples: the ***parameters*** — the atoms bound when the pattern is matched against the arguments, as discussed below — are unbound, but the LAMBDA term does not generate an error when processed. This is because neither neither the pattern nor the body is treated as an extensional argument. (Less clear, although hinted by S4-438, is the fact that the *type* argument to LAMBDA is processed at reduction time.)

Further evidence of this procedural intensionality with respect to the second and third argument position is provided in this example:

```
> ((LAMBDA EXPR [FUN]                                          (S4-442)
     (BLOCK (PRINT 'LAST) (FUN 1 2))
   (BLOCK (PRINT 'SHOE) +)) SHOE LAST
> 3
```

In other words processing of the argument occured *before* processing of the body of the λ-term. The body of a LAMBDA term, in other words, is processed each time the function it designates is applied. This fundamental fact about these expressions will motivate the

semantical account.

In spite of LAMBDA's intensionality, however, there is a sense in which the context of use of a LAMBDA reduction affects the behaviour of the resultant procedure when *it* is used. In particular, we have the following:

```
((LAMBDA EXPR [FUN]                                          (S4-443)
    ((LAMBDA EXPR [Y]
       (FUN Y))
     2))
 ((LAMBDA EXPR [Y]
    (LAMBDA EXPR [X] (+ X Y)))
  1))                                  ⇒  3
```

In other words, the atom FUN is bound to a function that adds 1 to its argument. This is because the Y in the body of the lexically last λ-term in the example (the second last line) receives its meaning from the context in which it was reduced (a context in which Y is bound to 1) not from the context in which the function it designates is applied (a context in which Y is bound to 2). In a dynamically scoped system, S4-443 would of course reduce to 4.

The expression in S4-443 is undeniably difficult to read. We will adopt a 2-LISP LET macro, similar to the 1-LISP macro of the same name, to abbreviate the use of embedded LAMBDA terms of this form (this LET will be defined in section 4.d.vii). In particular, expressions of the form

```
(LET [[<param1> <arg1>]                                      (S4-444)
      [<param2> <arg2>]
     ...
      [<paramk> <argk>]]
   <body>)
```

will expand into the corresponding expression

```
((LAMBDA EXPR [<param1> <param2> ... <paramk>]               (S4-445)
    <body>)
 <arg1> <arg2> ... <argk>)
```

Similarly, we will define a "sequential LET", called LET*, so that expressions of the form

```
(LET* [[<param1> <arg1>]                                     (S4-446)
       [<param2> <arg2>]
      ...
       [<paramk> <argk>]]
   <body>)
```

will expand into the corresponding expression

```
((LAMBDA EXPR <param1>                                          (S4-447)
    ((LAMBDA EXPR <param2>
       ...
         ((LAMBDA EXPR <paramk> <body>)
          <argk>))
       ...
      <ark2>))
 <arg1>)
```

Thus each $\langle arg_1 \rangle$ may depend on the bindings of the parameters before it. The difference between these two is illustrated in:

```
(LET [[X 1]]                                                    (S4-448)
   (LET [[X (+ X 1)]
         [Y (- X 1)]]
      Y))                          ⇒    0

(LET [[X 1]]                                                    (S4-449)
   (LET* [[X (+ X 1)]
          [Y (- X 1)]]
      Y))                          ⇒    1
```

Although some of the generality of LAMBDA is lost by using this abbreviation (all LETS and LET*s, for example, are assumed to be EXPR lambda's), we will employ LET and LET* forms rather widely in our examples. The expression in S4-443, for example, can be recast using LET as follows:

```
(LET [[FUN (LET [[Y 1]]                                         (S4-450)
             (LAMBDA EXPR [X] (+ X Y))]]
   (LET [[Y 2]] (FUN Y)))                        ⇒    3
```

The behaviour evidenced in S4-443 and again in S4-450 is of course evidence of what is called *static* or *lexical* scoping; if S4-443 reduced to the numeral 4 we would say that *dynamic* or *fluid* scoping was in effect. Dynamic and static scoping, however, are by and large described in terms of *mechanisms* and/or *behaviour*: one protocol is treated this way; the other that. It is not our policy to accept behavioural accounts — we are committed to being able to answer such questions as "*Why do these scoping regimens behave the way that they do?*" and "*Why was static scoping defined?*". Fortunately, the way we have come to look at this issue brings into the open a much deeper characterisation of what is happening. In particular, we said that LAMBDA was *intensional*, but this example makes it clear that it is not *hyper-intensional*, in the sense of treating its main argument — the body expression — purely as a structural or textual object. It is not the case, in other

words, that the application of the function bound to FUN in the third line of the example consists in the replacing, as a substitute for the word term "FUN" the textual object "(+ X Y)". To treat it so would yield an answer of 4 — would imply dynamic scoping. Rather, what is bound to FUN is neither the body as a textual entity, nor the *result* of processing the body, but rather something intermediate. It is an object closer to the *intension* of the body *at the point of original reduction.*

If we had an adequate theory of intensionality, it would be tempting to say that LAMBDA was a function from textual objects (the body expression and so forth) onto the *intension* of those textual objects in the context in force at the time of reduction. The subsequent use of such a function would then "reduce" (or "apply", or whatever intermediate term was chosen as proper to use for combining functions-in-intension with arguments) this intension with the appropriate arguments. Sadly, we have no such theory (furthermore, a somewhat more complex story has to be told: LAMBDA is of course a function from textual objects *onto functions*, as we made clear earlier; what we will show is that those functions *preserve the intension* of the textual argument). But the crucial point to realise here is that a statically scoped LAMBDA, which is what we have, is a *coarser-grained* intensional procedure than is a dynamically scoped LAMBDA.

> *Static scoping corresponds to an intensional abstraction operator; dynamic scoping to a hyper-intensional abstraction operator.*

In order to understand this claim in depth, we need to retreat a little from the rather behavioural view of LAMBDA that we have been presenting, and look more closely at what λ-abstraction consists in. It is all very well to show how LAMBDA terms behave, but we have not adequately answered the question "*What do they mean?*". They designate functions; that is clear. We know, furthermore, that functions are sets of ordered pairs, such that no two pairs coincide in their first element. We know what application is: a function *applied* to an argument is the second element of that ordered pair in the set whose first element is the argument.

However none of this elementary knowledge suggests any relationship between a function and a function *designator.* We do have a consensual intuition about λ — that it is an operator over a list of variables and expressions, designating the function that is signified by the λ-abstraction of the given variables in the expression that is its "body" argument.

However this intuition must arise independently, and therefore requires independent motivation and explanation. The fundamental intuition underlying LAMBDA terms, and λ-abstraction in general, can be traced back at least as far as Frege's study of predicates and sentences in natural language. A λ-term is in essence a *designator with a hole in it*, just as a predicate term is a *sentence* with a hole in it. If, for example, we take the sentence "*Mordecai was Esther's cousin*", and delete the first designating term, then we obtain the expression " ____ *was Esther's cousin*". It is easy to imagine constructing an infinite set of other derivative sentences from this fragment, by filling in the blank with a variety of other designating terms. Thus for example we might construct "*Aaron was Esther's cousin*" and "*the person who lives across the fjord was Esther's cousin*" and so forth. In general, some of these constructed sentences will be true, and some will be false. In the simplest case, also, the truth or falsity hinges not on the actual form of the designator, but on the *referent* of that designator. Thus our example sentence is true (at least so far as we know) just in case the supplied designator refers to Mordecai: any term codesignative with the proper name "Mordecai" would serve equally well.

Predicates arise naturally from a consideration of sentences containing blanks; the situation regarding designators — and the resultant functions — is entirely parallel. Thus if we take a complex noun phrase such as "*the country imediately to the south of Ethiopia*", and remove the final constituent noun phrase, we get the open phrase "*the country imediately to the south of* ____ ". Once again, by filling in the blank with any of an infinite set of possible noun phrases, the resultant composite noun phrase will (perhaps) designate another object. In those cases where the resultant phrase succeeds in picking out a unique referent, we say that the object so selected is in the *range* of the function, the object designated by the phrase we put into the blank is in the *domain*, and thus erect the entire notion of function with which we are so familiar.

Once the basic direction of this approach is admitted, a raft of questions arise. What happens, for example, if we construct *two* blanks? The answer, of course, is that we are led to a function of more than one argument. What if the noun phrase we wish to delete occurs more than once (as for example the term "*Ichabod*" in "*The first person to like Ichabod and Ichabod's horse*")? The power of the λ-calculus can be seen as a formal device to answer all of these various questions. The formal parameters are a method of labelling the holes: if one parameter occurs in more than one position within the body of the lambda

expression, then tokens of the formal parameters stand in place of a single designator that had more than one occurrence. If there is more than one formal parameter, then more than a single noun phrase position has been made "blank". And so on and so forth — all of this is familiar.

It is instructive to review this history, for it leads us to a particular stance on some otherwise difficult questions. Note for one thing how the function of LAMBDA as a pure naming operator becomes clear. In addition, it is important to recognise how *syntactic* a characterisation this is: we have talked almost completely about *signs* and *expressions*, even though we realised that the semantical import of the resultant sentence depended (in the simple extensional case) only on the referent of the ingredient noun phrase we inserted into the blank. The abstract notions of predicates, relationships, and functions were derivative on the original syntactic manoeuvering. Thus we have achieved a stance from which it is natural to ask essentially *syntactic* questions about the fundamental intuition (indeed, it is because we want the answers to syntactic questions that we are pursuing this line). For example, suppose we want to define a function, in some context, and do so by using some composite term into which we insert a blank. What, we may ask, is the natural context of use of that open sentence? If it is being used to define a function, then the only conceivable answer is that it is to be understood in the context where it is used to the define the function. Suppose, for example, that while writing this paragraph I utter the sentence "*Bob is going to vote for the President's oldest daughter*". Again staying with the simplest case, it is natural to assume that I refer to the President's oldest daughter, known by the name "Maureen Reagan". If I excise the noun "Bob" and construct the open sentence " ____ *is going to vote for the President's oldest daughter*", then I have constructed a predicate true of people who will vote for Maureen Reagan. This, at least, is the simplest and most straightforward reading. It is undeniably more complex, if nonetheless coherent, to suggest that we take the whole designator itself intensionally, so that *when we ask whether the resulant predicate is true of some person we will determine the referent of the phrase "the President" only at that point.* The ground intuition is unarguably extensional.

What does this suggest regarding LISP? Simply this: that the natural way to view lambda terms is as expressions that designate functions, where the function designated is determined with respect to the context of use where the lambda term is used ("used" in the sense of "stated" or "introduced" — not where the function it designates is applied). This

leads us to an adoption of statically scoped free variables, but only because we can show how that mechanism correctly captures this intuition. It is no accident that Church employed static scoping when he defined the λ-calculus: *static scoping is the truest formal reconstruction of the linguistic intuitions and practice upon which the notion of λ-abstraction is based.*

In order to remain true to Church's insight, then, we must be true to the understanding that his calculus embodies. There is no reason to propose a substitutional procedural regimen, for this would mimic his *mechanism*, rather than *what his mechanism was for.* It would be crazy for us to propose a substitutional reduction regime for 2-LISP — a formalism with procedural side-effects — since every occurence of a variable in a procedure would engender another version of the side-effects implied by the argument expression. This was not a problem for Church because he of course had no side-effects.

In sum, we will insist that the term

```
(LET [[Y 1]]                                                    (S4-461)
   (LAMBDA EXPR [X] (+ X Y))
```

designate the increment function, not that function that adds to its argument the referent of the sign "Y" in the context of use of the designating procedure.

As far as it goes, this is simple. We saw in chapter 2 how the static reading leads rather naturally to a higher-order dialect, to uniform processing of the expression in "function position" in a redex, and so forth, though we did not in that chapter examine the underlying semantical motivation for this particular choice. Nor did we examine explicitly a subject we must now consider: the *intensional* significance of a LAMBDA term.

That this last question remains open is seen when we realise that the preceding discussion argues only that the *extension* of the LAMBDA term be determined by the context of use in force at the point where the LAMBDA term was introduced. However it remains unexamined what role is played by the full computational significance of the term in "body" position — the open designator with demarcated blanks in it, to use our present reconstruction. In this regard it is instructive to look at the reduction regimen adopted by Church in the λ-calculus, which, as we have said, is a statically scoped higher order formalism. By the discussion just advanced, it should depend on an intensional LAMBDA, but of course no theory of functions-in-intension accompanies the λ-calculus. Nor is "λ", in the

λ-calculus, a *function*, since the λ-calculus is strictly an extensional system, and there is no way in which an appropriately *intensional* function could be defined within its boundaries. λ-terms in the λ-calculus, in fact, are demarcated *notationally*, as they were in the first version of 1-LISP we presented in chapter 2 (the lexical item "λ", in the lambda calculus, is on its own uninterpreted, like the left and right parentheses and the dot). The reduction regime, furthermore, is one of substitution, which would superficially appear to be a hyper-intensional kind of practice. *Actual textual expressions*, after all, are substituted one within another during the reduction of a complex λ-calculus term. The dictum a few pages back said that hyper-intensional abstraction corresponds to dynamic scoping (and intensional abstraction to lexical scoping). How then can we defend our claim of intensional abstraction in a statically scoped formalism?

The answer is that the λ-calculus is highly constrained in certain ways which enable hyper-intensional substitution protocols to mimic a more abstract intensional kind of behaviour. Two features contribute to this ability. First, there is no QUOTE primitive (and of course no corresponding disquotation mechanism), so that it is not possible in general and unpredictable ways to capture an expression from one context and to slip it into the course of the reduction in some other place "behind the back of the reduction rules", so to speak. Second, there is that very important rule having to do with variable capture, called α-reduction. It is a constraint on β-reduction — the main reductive rule in the calculus — that terms may not be substituted into positions in such a way that a variable would be "captured" by an encompassing λ-abstraction. If such a capture would arise, one must first rename the parameters involved in such a fashion that the capture is avoided. For example, the following is an incorrect series of β-reductions:

```
(λF.((λG.(λF.FG)) F))      ; This is an illegal derivation          (S4-452)
(λF.(λF.FF))               ; since this β-reduction is incorrect.
```

Rather, one must use an instance of α-reduction to rename the inner F so that the substitution of the binding of G for G will not inadvertently lead that substitution to "become" an instance of the inner binding. Thus the following is correct:

```
(λF.((λG.(λF.FG)) F))      ; This is a legal derivation             (S4-453)
(λF.((λG.(λH.HG)) F))      ; First we do an α-reduction, and
(λF.(λH.HF))               ; then a valid β-reduction.
```

The precise and only role of α-reduction in the λ-calculus is to re-arrange textual objects so as to avoid the dynamic scoping that would be implied if α-reduction did not exist.

The question we may ask, however, is *why* the reduction in S4-462 is ruled out — *why* dynamic scoping is so carefully avoided. The answer cannot be that the resulting system is incoherent, since β-reductions with no α-reductions is one way to view LISP 1.5 and all its descendents. Sure enough the Church-Rosser theorem would not hold, but, as LISPs have shown, one can therefore simply decide rather arbitrarily on one reduction order. But we now have an answer: it violates the claim that the formal apparatus retains the designation, attributed by the intuitive understanding of the significance of the original λ-term. More specifically, variable capture alters intension — thereby violating intention.

We have, then, the following result: the reduction of LAMBDA terms must, in a sense, *preserve the intension of the body expression.* This of course is a much stronger result than the overarching mandate that Ψ preserve *designation* in every case. Ψ, on the other hand, does *not* preserve intension generally, according to a common sense notion of intension. This is difficult to say formally, for two reasons, the most serious of which is that we don't have a theory of intension with respect to which to formulate it. If one takes the intension of an expression to be the function from possible worlds onto extensions of that expression in each possible world — the approach taken in possible world semantics and by such theorists as Montague[4] — then it emerges (if one believes that arithmetic is not contingent) that all designators of the same number are intensionally equivalent. Thus (+ 1 1) and (SQRT 4) would be considered intensionally equivalent to 2 (providing of course we are in a context in which SQRT designates the square-root function). It is the view of this author that this violates lay intuition — that a more adequate treatment of intensionality shoud be finer grained (perhaps of a sort suggested by Lewis[5]). Furthermore, without specifying the intensions of the primitive nominals in a LISP system, it is difficult to know whether intension is preserved in a reduction. Suppose, for example, that the atom PLANETS designates the sun's planets, and is bound to the rail [MERCURY VENUS EARTH ... PLUTO]. Then (CARDINALITY PLANETS) might reduce to the numeral 9 if CARDINALITY was defined in terms of LENGTH. It is argued that the phrases *"the number of planets"* and *"nine"* are intensionally distinct because *"the number of planets"* might have designated some other number, if there were a different number of planets, whereas *"nine"* necessarily designates the number nine in this language. On such an account the reduction of (CARDINALITY

PLANETS) to 9 is not intension preserving. But making this precise is not our present subject matter.

Furthermore, if all we ask of the reduction of LAMBDA terms to normal form is that intension be *preserved*, we do not have to *reify* intensions at all — we do not even have to take a position on whether intensions are *things*. All that we are bound to ensure is this: *that the intensional character of the expression over which the LAMBDA term abstracts be preserved in the function designator to which the LAMBDA term reduces.* At the declarative level this will be our guiding mandate.

However, with respect to LAMBDA terms we have a much more precise set of questions to answer, having to do with the relationship between the intensional content of a LISP expression and its computational significance. The issue is best introduced with an example that we will make use of later. It is a widely appreciated fact that, if an expression <X> should not be processed at a given time, but should be processed at another time, a standard technique is to wrap it in a procedure definition, and then to *reduce* it subsequently, rather than simply *using* it. A simple example is illustrated in the following two cases: in the first the (PRINT 'THERE) happens *before* the call to (PRINT 'IN); in the second it happens *after*.

```
> (LET [[X (PRINT 'THERE)]]                                    (S4-464)
     (BLOCK (PRINT 'IN) X)) THERE IN
> $T

> (LET [[X (LAMBDA EXPR [] (PRINT 'THERE))]]                   (S4-465)
     (BLOCK (PRINT 'IN) (X))) IN THERE
> $T
```

Because of 2-LISP static scoping (which corresponds to this intensional reading of LAMBDA), this approach can be used even if variables are involved:

```
> (LET* [[X 'THERE]                                            (S4-466)
         [Y (PRINT X)]]
     (BLOCK (PRINT 'IN) Y)) THERE IN
> $T

> (LET* [[X 'THERE]                                            (S4-467)
         [Y (LAMBDA EXPR [] (PRINT X))]]
     (BLOCK (PRINT 'IN) (Y))) IN THERE
> $T
```

What this example illustrates is that the side-effects engendered by a term (the input/output behaviour is illustrated here, but of course control and field effects are similar) take place

only when the term is processed in an extensional position. In other words if LAMBDA takes an intensional reading of the body expression, it does not thereby engender the full computational significance of that expression. Such significance arises only when some other function or context requires an *extensional* reading.

The QUOTE function in 2-LISP that we defined in S4-132, and handles in general, are hyper-intensional operators; it was clear in their situation that the significance of the mentioned term was not engendered by the reduction of the hyper-intensional operator over the term. We have not, however, previously been forced to ask the question of what happens with respect to *intensional* operators, but the examples just adduced yield an answer: they too do not release the potential significance of the term. It is for this reason that the "deferring" technique works in the way that it does. (Note that no suggestion is afforded by the λ-calculus with respect to this concern, since there are no side-effects at all.)

We have concluded, in other words, this constraint: *intension-preserving* term transformations do not engender the procedural consequences latent in an expression; those consequences emerge only during the normalistion of a redex, when *intension is not preserved.* Though (+ 2 3) reduces to *co-extensionsal* 5, it is on our view *not* the case that (+ 2 3) and 5 are intensionally equivalent.

We have one more question to examine before we can characterise the full significance of LAMBDA. In spite of our claim that LAMBDA is an intensional operator, it is not the case that LAMBDA is a function from expressions onto intensions, nor is it the case that LAMBDA terms reduce to intensions. If X is a term (LAMBDA ...), in other words, neither Φ(X) nor Ψ(X) is an intension. Both of these possibilities are rejected by protocols we have long since accepted. In particular, note that in any form (<F> . <A>), we have assumed that the significance of the whole arises from the application of the function *designated* by <F> to the arguments <A>. Thus in (+ 2 3), which is in reality (+ . [2 3]), we said that the whole designated five because the atom "+" designated the extensionalised addition function, which when applied to a syntactic designator of a sequence of two numbers, yielded their sum.

Similarly, in any expression

```
((LAMBDA <type> <params> <body>) . <args>)                    (S4-468)
```

it follows that the term (LAMBDA ...) must designate a function Similarly, in a construct like

```
(LET [[F (LAMBDA ... )]]
   (F . <args>))                                    (S4-469)
```

F must also designate a function. This is all consistent with our requirement that variable binding be designation preserving: F and (LAMBDA ...) must be co-designative.

It follows, then, that F cannot *designate* the intension of the (LAMBDA ...) term. Hence (LAMBDA ...) cannot normalise to a designator of that function's intension. For we do not know what intensions are, but they are presumably not syntactic, structural entities. They are not, in other words, elements of S, and Ψ has S as its range. We said earlier, however, that F must be intensionally similar to the LAMBDA term — what this brings out is that F must be ***co-intensional*** with the LAMBDA term, as well as *co-extensional.* The normalisation of LAMBDA terms, in other words, must *preserve* intension as well as extension.

This is as much as we will say regarding LAMBDA in its simple uses. In accord with our general approach, we have attempted to characterise LAMBDA terms primarily in terms of *what they mean*; from this we justified our account of *how they behave.* As usual, Ψ is subservient to Φ. It remains, finally, to remark on *how they work.* The answer to the latter question is of course quickly stated: when a LAMBDA reduction is itself processed, a closure (see below) is constructed and returned as the result. When a pair whose CAR normalises to a non-primitive closure is encountered, the closure is what we call *reduced* with the arguments. If that closure is an EXPR, then this reduction begins with the reduction of the CDR of the pair, followed by the binding of the variables in the parameter pattern against the resultant normal-form argument designator. If the closure is an IMPR, no argument normalisation is performed; instead a handle designating the CDR of the pair is matched against the parameter pattern. In either case the body of the closure (the body of the original reduction with LAMBDA) is processed in a context that, as usual, consists of a field and an environment. The field is the field that results from the processing of the arguments — as usual there is no structure to the use of fields: a single field is merely passed along throughout the computation. The environment, however, is this: it is the environment that was in force at the point when the closure was constructed, but augmented to include the bindings generated by the pattern match of arguments against variables.

it follows that the term (LAMBDA ...) must designate a function. Similarly, in a construct like

```
(LET [[F (LAMBDA ... )]]
   (F . <args>))                                    (S4-469)
```

F must also designate a function. This is all consistent with our requirement that variable binding be designation preserving: F and (LAMBDA ...) must be co-designative.

It follows, then, that F cannot *designate* the intension of the (LAMBDA ...) term. Hence (LAMBDA ...) cannot normalise to a designator of that function's intension. For we do not know what intensions are, but they are presumably not syntactic, structural entities. They are not, in other words, elements of S, and Ψ has S as its range. We said earlier, however, that F must be intensionally similar to the LAMBDA term — what this brings out is that F must be ***co-intensional*** with the LAMBDA term, as well as *co-extensional.* The normalisation of LAMBDA terms, in other words, must *preserve* intension as well as extension.

This is as much as we will say regarding LAMBDA in its simple uses. In accord with our general approach, we have attempted to characterise LAMBDA terms primarily in terms of *what they mean*; from this we justified our account of *how they behave.* As usual, Ψ is subservient to Φ. It remains, finally, to remark on *how they work.* The answer to the latter question is of course quickly stated: when a LAMBDA reduction is itself processed, a closure (see below) is constructed and returned as the result. When a pair whose CAR normalises to a non-primitive closure is encountered, the closure is what we call *reduced* with the arguments. If that closure is an EXPR, then this reduction begins with the reduction of the CDR of the pair, followed by the binding of the variables in the parameter pattern against the resultant normal-form argument designator. If the closure is an IMPR, no argument normalisation is performed; instead a handle designating the CDR of the pair is matched against the parameter pattern. In either case the body of the closure (the body of the original reduction with LAMBDA) is processed in a context that, as usual, consists of a field and an environment. The field is the field that results from the processing of the arguments — as usual there is no structure to the use of fields: a single field is merely passed along throughout the computation. The environment, however, is this: it is the environment that was in force at the point when the closure was constructed, but augmented to include the bindings generated by the pattern match of arguments against variables.

If we were equipped with a theory of functions in intension, and could avail ourselves of an intensional operator in the meta-language, called INTENSION-OF, that mapped terms and lists of formal parameters into *intensions* — whatever they might be — we could specify the desired semantical import of LAMBDA in its terms. But, lacking such a theory, we will instead look at LAMBDA from the point of view of designation and reduction, armed with the mandate that it is the intensional properties of the resultant structures that are of primary concern.

4.c.ii. Closures: Normal-form Function Designators

Two questions press for resolution. First, since we do not have the theory of intensionality called for in the previous section, we need to formulate an alternative account of LAMBDA's semantics. Secondly, we need to answer a question we have side-stepped numerous times in this chapter: what is the form of normal-form function designators? Our over-arching normalisation mandate requires of us that expressions of the form (LAMBDA ...) normalise to a term that meets the constraints on normal-formedness, and designate the function designated by the LAMDBA term. We said in section 4.a.vii that we would use pairs as the structural category for such terms; we said in section 3.f.ii that we would employ the normal form designator of the EXPR function as the structural item in functional position. Section 4.c.i argued that normal-form function designators should be intensionally equivalent to the LAMBDA tems from which they arise. Finally, we said that we would define as a ***closure*** any term that meets these various conditions. We need to examine just what 2-LISP closures are.

One purpose of the discussion in the immediately preceding section, among others, was to convey as sense of what closures must do. We wanted them to encode within themselves the identity of the intension of the funtion designated, which, as we pointed out, was some function of the context of use and the textual term in body position. But, when put this way, the answer is clear: if we know that the intension is a function of these two things, then if we *store* those two things (or store informationally complete designators of them) we are guaranteed to have preserved sufficient information to reconstruct the context and LAMBDA term originally employed. Also, if we know how to move in a single step from textual item plus context plus arguments to the full reduction, then if we have preserved the entire context when we wish to apply/reduce the intension we can pretend we are working in the standard extensional situation. In other words, though we don't know how to reify intensions, we can be sure we have preserved the proper intensional properties if we can back up to an equivalent hyper-intensional form plus context, and, so to speak, "come back through again".

This is why closures contain encodings of environments. If we had a theory of intension we would not need to define them in this fashion, but for the time being this

approach must suffice. It is rather inelegant, as the reader should be aware, for the following reason, among others: environments, as we have been at pains to say again and again, are theoretical posits with which we have made sense of LISP's behaviour: never before have environments entered into our actual domain of discourse. What we said in section 3.f.iii bears repeating: environments have up until this point been objects *in the semantical domain of the theoretical meta-language*, not in either *S* or *D*. However, our lack of a theory of functions-in-intension forces us to have closures encode environments within them: this is the meaning of the underlined $\underline{E_0}$ term that occurs as the first argument to EXPR in all of the closures presented throughout the earlier parts of this chapter. In other words, against all of our methodological principles, the object-level structure of the 2-LISP language will be theory-relative (thus fundamentally challenging our operating assumption that a higher-order meta-structural dialect can be obtained in a theory-free fashion).

In 3-LISP this encoding of environments within closures is not quite as inelegant as in 2-LISP, because structurally encoded theoretical accounts of the processor play a major role. However even there there remains a slight inelegance — the shadow of the same lack that plagues us here. The notion *environment*, being a term in a theory of LISP, should enter the discussion as a word that is *used* at a meta-level. This is the case when environments (along with continuations) are bound to variables by reflective procedures. However environments also enter into closures at the object level, as they do here in 2-LISP, and as they properly ought not to do. Thus even 3-LISP would be cleaner if a computable and finitely representable intensional object were forthcoming. (On the other hand, it should be admitted that the inclusion of structural environment designators within closures will prove extremely convenient when we discuss the question of *changing* a closure to designate a different function, in accord with new definitions of constituent functions. Thus this theory-relative encoding has its apparent advantages. It is not, however, possible to argue at this time that a more adequate intensional encoding would not provide similar benefits.)

The form of a closure, then, will be this:

(<EXPR> <ENV-D> <PATTERN> <BODY>) (S4-465)

where <EXPR> is the EXPR closure, <ENV-D> is an environment designator, <PATTERN> designates the parameter pattern, and <BODY> designates the body. For both consistency

and elegance, in other words, we have chosen to have *all* the arguments to <EXPR> designate the closure ingredients; in this way <EXPR> can be itself an EXPR. It enables us, furthermore, to have as normal-form redexes only those <EXPR> redexes whose arguments are themselves in normal-form. Thus both of the "pseudo-composite" structural types — pairs and rails — will be in normal-form only if their "ingredients" are in normal form (although there is an asymettry in the other direction: *any* rail whose elements are in normal-form is by definition itself in normal form, whereas not every pair whose CAR and CDR are normal is itself normal).

Since the second and third arguments to <EXPR> designate structures, they will in the normal-form case necessarily be handles. Thus we would expect:

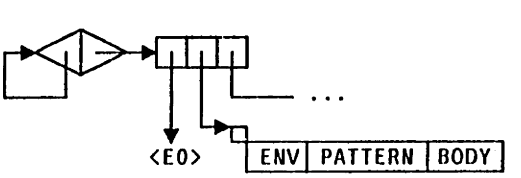
```
(LAMBDA EXPR [N] (+ N 1))      ⇒    (<EXPR> ? '[N] '(+ N 1))      (S4-466)
```

The question regarding the structure of environment designators was answered in section 3.f.ii: since environments are sets of ordered pairs of atoms and bindings, environment *designators* are rails consisting entirely of two-element rails, with each sub-rail consisting of two handles; the first designating the atom, and the second designating the binding. Thus the general environment designator will be of the form:

```
[['<ATOM1> '<BINDING1>]                                    (S4-467)
 ['<ATOM2> '<BINDING2>]
  ...
 ['<ATOMk> '<BINDINGk>]]
```

Two questions remain, about what environments are actually in force, and about the form of <EXPR>. The first will be answered only in section 4.c.vi, when we take up global bindings, top-level definitions, and SET. The second was sketched in section 3.f.ii; we said there that the atom EXPR would be bound, in the initial environment, to a closure of the following structure (this is the straightforward 2-LISP translation of the 1-LISP structure pictured in S3-200):

(S4-468)



However we can fill this out now more explicitly. We first give an admittedly circular definition of EXPR:

```
(DEFINE EXPR                                                    (S4-469)
   (LAMBDA EXPR [ENV PATTERN BODY]
      (EXPR ENV PATTERN BODY)))
```

There is however a difficulty — or perhaps more accurately — an *incompleteness* here. Closures are in normal-form; therefore they are self-normalising, a fact that is determined primitively by the processor. Thus we have:

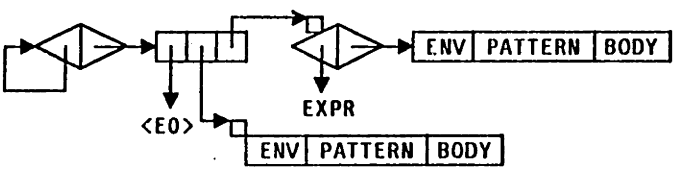
```
(LET [[X ↑(LAMBDA EXPR [N] (+ N 1))]]                           (S4-470)
   (= (NORMALISE X) (NORMALISE X)))          ⇒    $T
```

which is not predicted by S4-469. Thus the self-normalising aspect of normal-form expressions must be considered as prior to, and not captured in, the definition just given. Nonetheless, for other purposes S4-469 is adequate, implying that the EXPR closure would be of this form:

```
EXPR   ⇒   (<EXPR> E0                                           (S4-471)
                   '[ENV PATTERN BODY]
                   '(EXPR ENV PATTERN BODY))
```

as illustrated in the following graphical notation:

(S4-472)

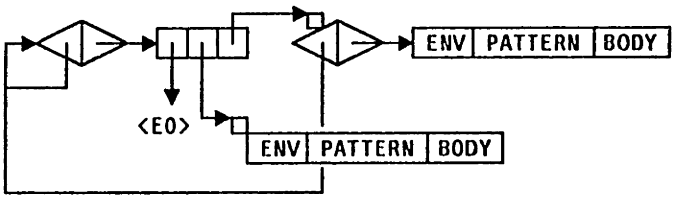


It is truer to the primitive nature of this closure, however, to avoid the explicit reduction of EXPR in the function position of the recursive (circular) call to EXPR; this more clearly suggests the normal-formedness of this form. Thus we will assume the following primitive <EXPR> structure:

```
EXPR   ⇒   (<EXPR> E0                                           (S4-473)
                   '[ENV PATTERN BODY]
                   '(<EXPR> ENV PATTERN BODY))
```

again as illustrated in graphical notation:

(S4-474)



Given this characterisation of EXPR, we need to look again at LAMBDA. We said earlier that LAMBDA, though procedurally intensional, was nonetheless extensional with respect to its first argument. The problem with presenting a definition, even circular, of this procedure is that it must do something without precedent: it must somehow reach into the workings of the processor and extract a true designator of the environment in force at the point of reduction. There being no mechanisms for this, we will instead present a (circular) definition in pseudo-3-LISP, for the *result* is the same — the difference is merely that in 3-LISP the mechanisms by which the result is obtained are mechanisms provided to the user. We have, in particular, the following (the up and down arrows can be ignored for the present; they merely mediate between the reflected level and the fact that the closure must itself be an object level expression):

```
(DEFINE LAMBDA                                          ; 3-LISPish      (S4-475)
   (LAMBDA REFLECT [[TYPE PATTERN BODY] ENV CONT]
      (CONT ↑(↓(NORMALISE TYPE ENV ID) ENV PATTERN BODY)))
```

This definition leads us to an examination of the role of the first argument to LAMBDA. In every example we have used so far, we have used "EXPR" or "IMPR" almost as if they were keywords selecting between simple extensional and intensional procedures. It is clear, however, that this argument position plays a potentially much larger role in determining the significance of a LAMBDA term. Our approach, furthermore, means that no keywords are needed, and facilitates the use of other functions in this position. A striking example where this power is used is in the 3-LISP definition of MACRO. In that dialect we will be able to define a function called MACRO to support such definitions as:

```
(DEFINE INCREMENT                                                        (S4-476)
   (LAMBDA MACRO [X] `(+ ,X 1))
```

with the consequence that the normalisation of the form

```
(INCREMENT (* X Y))                                                      (S4-477)
```

will engender the subsequent normalisation of the explicitly constructed expression

```
(+ (* X Y) 1)                                                    (S4-478)
```

The definition in S4-475 shows how this will proceed. The normalisation of INCREMENT will lead to the normalisation of

```
(MACRO <ENV> '[X] '`(+ ,X 1))                                    (S4-479)
```

Though we do not have enough machinery to define a suitable MACRO yet, its job is clear: it must yield an intensional closure such that when reduced, that closure will construct and normalise the appropriately instantiated version of the schematic expression given as the body in S4-476.

We will not pursue any uses of the type argument to LAMBDA in this chapter; the definition of MACRO, and other extensions, will be examined in chapter 5.

Finally, we should inquire about IMPRs. Strikingly, the IMPR closure is almost identical to the EXPR closure, although whereas EXPR was an EXPR; IMPR is not an IMPR: IMPR is also an EXPR. In particular, we have this approximate definition:

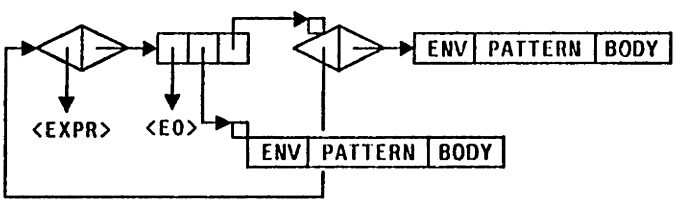
```
(DEFINE IMPR                                                     (S4-480)
   (LAMBDA EXPR [ENV PATTERN BODY]
      (IMPR ENV PATTERN BODY)))
```

and this structure to the primitive IMPR closure:

```
IMPR   ⇒   (<EXPR> E0                                            (S4-481)
                   '[ENV PATTERN BODY]
                   '(<IMPR> ENV PATTERN BODY))
```

as illustrated in graphical notation:

(S4-482)



Note that since <IMPR> is an EXPR, the body of the IMPR closure is not an intensional redex — which would be declaratively wrong.

One final comment needs to be made before we turn to characterising the semantics of LAMBDA, EXPR, and IMPR more carefully. The inclusion of the environment within a

closure interacts with the ability of SET to modify environments in force. This topic will be pursued in greater length in section 4.c.iv, but it is worth mentioning here. In particular, suppose that some LAMBDA body uses variable X freely: then the binding of X will be that of the environment in force when the LAMBDA term was reduced. Subsequent changes to that variable, in virtue of SET, may potentially modify the closed environment. Thus for example we have the following behaviour:

```
(LET* [[X 3]                                                    (S4-483)
       [F (LAMBDA EXPR [Y] (+ Y X))]]
      (BLOCK (SET X 4)
             (F 2)))                          ⇒  6
```

F was originally bound to a closure designating a function that adds three to its argument; the SET, however, since it affects the environment in which the LAMBDA term is closed, modifies the binding within the closure as well.

It is at least arguable that this is not always the behaviour one desires. Our analysis in terms of intension explains why: *if* we could map LAMBDA terms onto more stable intension encodings, then the binding of X at the point of reduction of the LAMBDA term would hold independent of *subsequent* alterations to that environment. It is for this reason that some dialects (INTERLISP and SEUS are examples) allow one to specify, through some other mechanism, those variables over which a LAMBDA term should be closed, in such a fashion that subsequent alterations to the binding of that variable do not affect the closure itself. What our present analysis has shown us is how this vulnerability to the subsequent modification arises out of our lack of an adequate intension operator. However in our own defense we should add that we will be able to define (in section 4.c.vi) a straightforward utility procedure that will facilitiate the construction of closures that explicitly protect themselves from the effects of subsequent modifications to the variables used freely within them.

The ability to modify the function designated by a closure (strictly, to change what function a closure designates by changing the closure itself — there is no meaning to the notion of actually changing a function) will prove useful in reflective work. We said earlier that SET is not a primitive in 3-LISP; instead, it is defined as a reflective procedure that wreaks side-effects on environment designators. It can as well wreak side-effects on closures, thus altering what functions they designate. This ability is important to provide — an example is the ability to redefine procedures used by closures, which is critical in

debugging and program development. However we will argue that it should not be *confused* with the normal use of SET in the object level of a program.

4.c.iii. Parameter Patterns and Variable Binding

In the original intuition, λ-abstraction uses a single formal parameter to mark the hole or holes in the composite designator. Thus, in the expression

```
λX.((FX)(GX))                                                    (S4-484)
```

a single X marks two occurences of the same hole — a hole, in other words, to be filled by two occurences of the same designator. We said above that multiple arguments — holes to be filled by *different* designators — arise in a natural way, but there are a variety of formal mechanisms we could use to implement them. For example, if we consider addition, and use as our source template the term (+ 3 4), we could abstract this, over both the "3" and "4" positions, in any of the following ways:

```
(LAMBDA EXPR Z (+ (1ST Z) (2ND Z))                               (S4-485)

(LAMBDA EXPR A (LAMBDA EXPR B (+ A B)))                          (S4-486)

(LAMBDA EXPR [A B] (+ A B))                                      (S4-487)
```

In the first we have reconstituted the template expression, so that only a single hole remains (although there are two occurences of it); in this way we can retain the machinery that accepts a single argument. In the second we use two separate abstractions, one for each blank. Thus the first abstracts the "3" position, and the second abstracts the "4" position. This is the "currying" approach, mentioned earlier, that we use in our meta-language. In the third we apparently extend our syntactic mechanism to support two arguments in a seemingly obvious way.

As discussed in section 4.a.v, the second approach fundamentally conflicts with our objectification mandate, in spite of its formal generality. At first blush the third would seem to do so as well, since it conveys the impression that a procedure defined in this way would have to be called with exactly two argument expressions. Thus it would appear that the objectification mandate would force us to adopt the first of the three suggestions. On the other hand the third candidate is manifestly the most convenient — a fact to which immediate intuition and standard LISP practice both attest. It remains to be explained, however, what a *rail* of two *atoms* in a parameter position of a LAMBDA term *means*.

A little investigation will show us that we can adopt the third candidate *syntactically*, while making it *semantically* like the first. The approach emerges from the realisation that the binding of variables or formal parameters is an extremely simple case of pattern matching. We have already said that every 2-LISP (and 3-LISP) function/procedure is called with a single argument — this was made clear as early as section 4.a.iv. In those cases where the natural conception is of a function applied to multiple arguments, the function will in fact be applied to a single sequence — an abstract mathematical ordered set — of arguments instead. The parameter structure in a LAMBDA term, however, will be allowed to be built up out of atoms and rails. Thus we will encounter such LAMBDA terms as:

```
(LAMBDA EXPR ARGS ... )                                    (S4-488)
(LAMBDA EXPR [A B C] ... )
(LAMBDA EXPR [[X] Y [[Z W R]]] ... )
```

We will call the entire parameter structure the ***parameter pattern*** or ***pattern***; the atoms within it will be the ***parameters*** themselves. It is of course only the parameters that are bound; no sense is to be made of binding a rail. Nonetheless, the pattern as a whole determines how the parameters are bound, given a particular designated argument. The general mandate governing the binding — a mandate we will call the ***schematic designator*** principle — is this:

> *The pattern, if used as a designator in the environment resulting from the binding of a procedure's formal parameters, should designate the full argument to which the function is applied.*

This mandate is of course satisfied by the paradigmatic single argument case. In particular, if some function F was designated by the λ-term

```
λX.GX                                                      (S4-489)
```

and F was reduced with some other expression — say, (+ 1 2) — then we would expect the parameter X to be bound to the numeral 3. Thus a subsequent use of the term X would designate the number three, which is just what (+ 1 2) designates. Suppose, to extend this to a multi-argument case, that we had instead the more complex function designator (we switch to 2-LISP)

```
(LAMBDA EXPR [X Y] (G X (H Y)))                                    (S4-490)
```

bound to F, and this was used as follows:

```
(F (+ 1 2) 4)                                                      (S4-491)
```

then, since S4-491 is a lexical abbreviation for

```
(F . [(+ 1 2) 4])                                                  (S4-492)
```

F would be applied to a sequence of the two numbers (three and four). The only normal-form bindings of X and Y such that [X Y] would designate this sequence are of course that X be bound to the numeral 3 and Y bound to the numeral 4. That bindings be to *normal-form* designators is mandated by the fact that F is an EXPR, of course, although, as we will discuss later, even IMPRS (and 3-LISP REFLECTS) receive their bindings in normal form.

Thus the parameter pattern may, to use popular terminology, "de-structure" sequences of arguments. The fact that it is the *designated sequence* that drives the de-structuring, not the *structure of the argument designators*, grants us just the freedom we wanted to enable us to use non-rail CDRS without colliding with the binding mechanism, as for example in the expression

```
(+ . (REST [10 20 30]))                                            (S4-493)
```

Furthermore, it adequately treats what in MACLISP are called LEXPRS (INTERLISP "no-spreads"). It should be clear just why this freedom arises:

> *The relationship between argument structures and parameter structures in extensional procedures has only to do with designation; no formal relationships between the two are of any consequence.*

This, at least, is the overarching constraint. Because of the intension-preserving aspects of the binding of parameters to normal-form argument expressions, this is in some cases violated, but we can still use it to define the principal protocols, around which other developments will be organised.

It is of course both simple and elegant to enable this de-structuring to recurse: thus we could have:

```
((LAMBDA EXPR [[A B] [C D]]                                        (S4-494)
   (+ (* A C) (* B D)))
 (REST [10 20 30])
 (REST [5 15 25]))                  ⇒    1050
```

It is sometimes thought that the formal parameter section of a procedure consists merely of a "list of variables". It is instructive to contrast this view with the one we have adopted. First, a "list of variables" would in 2-LISP be represented as, to take S4-490 as our example, as

```
['A 'B]                                                    (S4-495)
```

But there is of course something odd about this. We have admitted that the "parameter pattern" argument position to LAMBDA is inherently intensional; thus it is arguable that it should be possible to omit the explicit quotation implied in S4-495. Ignoring for a moment the rail/sequence distinction, we could then allow [A B] in place of S4-495. On its own, however, this doesn't answer a number of crucial questions; it would have to be added explicitly that the order of parameters should match the order of arguments. Nor does it explicitly admit of recursion, or facilitate the use of a single atom parameter when it is desired to obtain a name designating the entire argument sequence. In this "list of variables" approach all of these complexities would require private explication and specification; the schematic designator mandate, however, couped with the fact that all 2-LISP procedures are semantically called with a single argument, answers them in one sweep.

There are a variety of questions that arise in any multiple argument scheme. We have not explained the significance of multiple occurences of the same atom in the parameter pattern, for example (in (LAMBDA EXPR [X Y X] ...), for example). We also need to indicate the consequences of calling a procedure with a sequence that is longer than that potentially designated by the parameter pattern, as illustrated for example in

```
((LAMBDA EXPR [X Y] (+ X Y))                                   (S4-496)
 1 2 3)                          ⇒     ???
```

Again, the schematic designator mandate supplies answers. In the former case, the pattern should match if and only if the first and third argument of the sequence are identical. The latter suggestion is ruled out; no binding of X or Y can render "[X Y]" a designator of the sequence of the first three natural numbers.

This pattern matching binding protocol is of course not new in its surface form, but it is instructive to follow out just a little the consequences of the semantical way we have defined it. Note as well that we have in 2-LISP six structural types, of which only two have been mentioned in the foregoing discussion. We bind only atoms; this is a decision that

was long ago fixed in the dialect. However it does not follow from that fact that only atoms may occur in patterns, as the rail examples have made clear. It is therefore worth exploring what would be implied by occurences of other structural forms in parameter patterns, given the mandate just laid out.

There are in particular four categories of structural object to be considered, of which three (booleans, handles, and numerals) are *constants*, in the sense that they designate their referents independent of context. Thus if one of them were to appear in a pattern, the governing mandate could apply only in case their referent was the very semantical entity in the designated argument. For example, the mandate could be satisfied in a reduction of the following form

```
((LAMBDA EXPR [X 3 Y $F] 'OK)                                    (S4-497)
 (* 1 2) (+ 1 2) (- 1 2) (= 1 2))        ⇒     'OK
```

but is impossible in this case:

```
((LAMBDA EXPR ['X X]) 3 4)                                       (S4-498)
```

because 'X designates the atom X, and can in no environment designate the number three, as would be required in order for this to be meaningful.

The ability to use constants in parameter patterns is probably not useful in a serial language. If 2-LISP were a pattern-matching formalism, so that at a given step in the course of a computation a variety of procedures could potentially be reduced, with the choice based on the possible match between their parameter patterns and the argument structures, then such a facility would be of interest. Such a calculus, however, might want additional facilities, so that *two* occurences of a single variable could be treated in the obvious fashion. We might want, for example, the reduction of

```
((LAMBDA EXPR [X Y X] (/ X Y))                                   (S4-499)
 (* 2 2) (- 2 2) (+ 2 2))
```

to bind X to 4 and Y to 0, yielding 0 as a result. The following, however, should fail:

```
((LAMBDA EXPR [X X] 'OK) 3 4)                                    (S4-500)
```

But this is a different language. While admitting the possible extension of our dialects in such a direction, we will not adopt these suggestions here. Thus the three constant structural categories will for the present be ruled illegal in parameter patterns.

Of much more interest is the use of pairs — of reductions — in a pattern. First, it is of course a consequence of the separation of pairs and rails that the question is open, even though we have admitted arbitrary de-structuring by rails. In standard LISPs, debate has arisen over complicating the binding protocols, illustrated by the following examples. One school has argued for destructuring similar to the rail proposal we have adopted: thus in such a proposed LISP,

```
(LET (((A . B) (CONS 3 4)))          ; This is not 2-LISP          (S4-501)
   <BODY>)
```

would bind A to 3 and B to four. The opponents have suggested on the contrary that non-atomic structures in binding position be treated rather like the intensional functions we saw in SETF in chapter 2; thus in

```
(LET (((CAR B) (+ 2 3)))             ; This is not 2-LISP          (S4-502)
   <BODY>)
```

either the CAR of B would be bound to 5, or else the CAR of B would be *made* 5 (implying a RPLACA) (we will consider these two possibilities in a moment) In 2-LISP we of course have approximately both options. The first (S4-501) would result in:

```
(LET [[[X Y] (SCONS 3 4)]]                                          (S4-503)
   <BODY>)
```

whereas the second (S4-502), should we decide to support it, would look instead like:

```
(LET [[(CAR B) (+ 2 3)]]             ; This is not 2-LISP yet      (S4-504)
   <BODY>)
```

The question, then, is what sort of sense to make of this last proposal.

The schematic designator mandate provides a strong guiding principle. Two examples in particular illustrate its force. Suppose first that we had a procedure F defined as follows:

```
(DEFINE F (LAMBDA EXPR (PREP X Y) <BODY>))                          (S4-505)
```

and we used it as follows:

```
(F 10 20 30)                                                        (S4-506)
```

The principle requires this: that X and Y be bound so that (PREP X Y) *designate* the mathematical sequence <10, 20, 30>. No mention is made of other computational significance of (PREP X Y); thus we are free to ignore (for the moment) the fact that it

would generate an otherwise-inaccesible structure if processed. We are required, as well, to ensure that X and Y are bound to normal-form designators. Thus X should clearly be bound to the numeral 10 and Y to the rail [20 30]. In other words we have a method (should we be able to generalise it sufficiently so that it warrants adoption) whereby such MACLISP expressions as

```
(DEFUN F (X &REST Y) <BODY>)          ; This is MACLISP          (S4-507)
```

fall out of the basic structure of the dialect, without requiring the addition of keywords or other extraneous language elements.

Another example has to do with the level-crossing primitives NAME and REFERENT. We said above that a function designator F of the form

```
(LAMBDA EXPR [X 'Y] <BODY>)                                      (S4-508)
```

would be ruled out, since 'Y can only designate Y. If however, we used instead

```
(LAMBDA EXPR [X ↑Y] <BODY>)                                      (S4-509)
```

which is an abbreivation for

```
(LAMBDA EXPR [X (NAME Y)] <BODY>)                                (S4-510)
```

then our governing mandate requires that the atom Y be bound to some normal-form designator such that (NAME Y) designate the argument. Thus if we used

```
(F '3 '4)                                                        (S4-511)
```

and X were bound to the handle '3 and Y was bound to the *numeral* 4, then [X (NAME Y)] would be equivalent to ['3 (NAME 4)], which would in turn be equivalent to ['3 '4], as required. Similarly, if G were defined as

```
(LAMBDA EXPR [X ↓Y] <BODY>)                                      (S4-512)
```

and used in

```
(G 3 4)                                                          (S4-513)
```

then X would be bound to the numeral 3 and Y to the *handle* '4, since ↓'4 designates the number four.

Such facilities could be of use, although a variety of cautions need to be kept in mind. For example, none of 'X, ↑X, and ↓X imply that X be bound *to the un-normalised argument structure* (or bound to a designator of the un-normalised argument structure), as if

a mechanism had been discovered so that intermediate procedures between EXPRs and IMPRs could be defined. IMPRs (and in 3-LISP, reflective procedures) still need to be employed for such purposes. Nor is it in general computable how to assign the open parameters in an arbitrary expression <X> so as to ensure that <X> designate a given semantical entity. Unification algorithms restricted so that only terms in one of the two expressions may be expanded could be used, but there are severe limits on such an approach. It is likely that even a moderate step in this direction would unleash virtually all of the problems associated with unification protocols, pattern-directed computation, and the like.

For present purposes, therefore, we will reject the suggestions just presented. 2-LISP and 3-LISP parameter patterns will be constrained to consist only of arbitrary combinations of rails and atoms.

There are three final comments to be made about parameter binding. First, it might seem that by introducing even a very mild version of pattern matching into the binding of formal parameters we have unleashed a raft of potential complications that could have been avoided had we used instead a more traditional "list of variables" approach. However *any* binding protocol is in its own small way a pattern matcher. Merely the question of whether the procedure has been called with the correct number of arguments, for example, is in essence a question of the "fit" or "match" between the parameter structure and the argument structure. Similarly, type-checking in typed languages involves a pseudo-semantic, rather than purely structural, version of matching. It is our intent not to introduce an otherwise absent notion, but rather to capitalise on the concepts that underly parameter binding in the general case.

The second comment is this: the discussion just given, in line with our general approach, specifies binding protocols semantically, rather than in terms of implementable behaviour. However it is clear that there is a very natural resonance between the normalisation of sequence designators and the use of rails in parameter patterns. In particular, we specified in section 4.a that rails were the normal-form designators of sequences. Given the semantical type theorem, we know that if we normalise any 2-LISP sequence designator successfully, we will obtain a rail. It is then a straightforward task, in terms of computational complexity, to match such a rail against a parameter structure, given one proviso: that no single parameter occur more than once within the parameter structure.

At any step, we simply need to check whether the paramater is an atom; if it is, we bind the whole normal-form argument designator to that atom; if it is not, the pattern must be a rail, and we recursively match each element of the rail against each element of the argument rail, checking only that they are of the same length.

Third and finally, in a major concession to pragmatics, we will adopt one extension that violates the schematic designator mandate endorsed earlier. It turns out that in using 2-LISP it is very often the case that, given a rail R, one wants to bind parameters to designate its elements (MACRO and IMPR procedures are typical cases, but there are others as well). For example, consider the intensional redex

```
(TEST A B (F C D))                                    (S4-514)
```

This is of course an abbreviation of

```
(TEST . [A B (F C D)])                                (S4-515)
```

If TEST is an intensional procedure, defined as follows

```
(DEFINE TEST                                          (S4-516)
   (LAMBDA IMPR [ARG1 ARG2 ARG3] ... ))
```

we have assumed throughout that we could asume, on processing S4-514, that ARG1 will designate A, ARG2 will designate B, and ARG3 will designate (F C D). However the "schematic designator" mandate of course fails: the argument expression is the *rail* [A B (F C D)]; the only possible parameter pattern that could designate it is a single ATOM — say, ARGS — with the result that ARGS would be bound to the *handle* '[A B (F C D)]. What we intend, however, is that ARG1 be bound to the *handle* 'A, ARG2 to the handle 'B, and ARG3 to the handle '(F C D).

If rails were sequences, then this result would follow automatically. In other words, if we could view rails simply as sequences of structures, rather than more particular *rails* of structures, then we would be able to engage in this sort of practice without extending our matching protocols. Since rails are *not* sequences, however, but since this kind of binding is nonetheless useful, we will adopt the following extension to parameter matching: a *rail* of sub-patterns in a pattern will match an argument expression if the parameters can be bound in such a way as to designate the referent of the argument expression, or to designate the sequence of elements of a rail, should the referent of the argument expressions be a rail.

Though this is undoubtedly a concession, note that it does not violate semantic level. The bindings that it allows — those facilitating S4-514, for example — would be generated by the simpler mandate if a *designator of a rail of structures* were equivalent to a *rail of designators of structures.* The extension we are adopting, in other words, essentially allows the "referent-of" and "element-of" operators to *commute*, which strictly they do not. It does not allow one of them to be by-passed, which would be considerably less acceptable. In addition, it is compatible in spirit with the use of NTH and LENGTH — paradigmatically operators on sequences — over rails as well. It was this original extension that led us to the definition of the semantic type *vector* in section 4.b.vi. Thus one way to describe this extension is this: just for the purpose of matching, a (schematic) rail may be viewed as a designator of a vector of either type.

In order at least to be symmetric, we should enable rails of designators to be taken as designators of rails, as well as the other way around. This extension — this backwards commuting of the same two predicates — also proves extraordinarily useful, adding practical force to the argument for it. In particular, we will find it convenient to allow ↓<X>, *if* <X> *is a sequence of designators,* to designate a sequence of the elements designated. Again this is a pure *extension*, in the sense that the domain of the "reference" function is being slightly extended beyond *s* to include sequences of elements of *s*. For example, we will allow an expression such as:

```
↓['2 '3 '4]                                    (S4-517)
```

to normalise to this:

```
[2 3 4]                                        (S4-518)
```

Without the convention S4-517 would be semantically ill-formed.

It would of course be possible to avoid this extension entirely, and still support the desired behaviour, if we *identified* rails of normal-form structure designators (i.e. rails of handles) with sequences — if, in other words, we accepted mathematical identity conditions on these (or indeed on all) rails. We will not pursue this suggestion here, however, since it would change in a considerable measure the kinds of structural field modifications we would allow.

Our new matching protocol, then, is effected by the following procedure (taken from the meta-circular processor in section 4.d.vii):

```
(DEFINE MATCH                                                    (S4-519)
  (LAMBDA EXPR [PATTERN ARGS]
    (COND [(ATOM PATTERN) [[PATTERN ARGS]]]
          [(HANDLE ARGS) (MATCH PATTERN (MAP NAME ↓ARGS))]
          [(AND (EMPTY PATTERN) (EMPTY ARGS)) (RCONS)]
          [(EMPTY PATTERN) (ERROR "Too many args supplied")]
          [(EMPTY ARGS) (ERROR "Too few args supplied")]
          [$T (JOIN (MATCH (1ST PATTERN) (1ST ARGS))
                    (MATCH (REST PATTERN) (REST ARGS)))]])))
```

Though we will not mention this extension widely, it will be used in many of our reflective and pre-reflective examples, particularly in section 4.d and in chapter 5.

4.c.iv. The Semantics of LAMBDA, EXPR, *and* IMPR

We turn in this section to a formal characterisation of the semantics of LAMBDA, EXPR, and IMPR. It should be realised that we do this before we have considered recursion or definitions. That we can do so is an important to recognise: the subtleties that come up with more complex naming interactions are to a certain extent external to the notion of LAMBDA abstraction itself; they are better considered as questions about the *use* of LAMBDA abstraction, as the subsequent discussion will make clear.

If all 2-LISP procedures were EXPRs, the definition of LAMBDA would be straightforward. We assume a function ENV in the meta-language that returns a normal-form designator of an environment; thus ENV is a function of type *[[ENVS × FIELDS] → S]*. We would have the following full significance of the primitive LAMBDA:

```
Σ[E0("LAMBDA)]                                                    (S4-520)
   = λE.λF.λC.
       C("(IMPR E0 [PARAM BODY] (LAMBDA PARAM BODY)),
         [λS1.λE1.λF1 .
            [λ<S2,E2,F2> .
                 Σ(S2,E2,F2,
                   [λ<S3,D3,E3,F3> .
                        Σ(NTH(2,S1,F3),EXTEND(E1,NTH(1,S1,F3),S3),F3,
                          [λ<S4,D4,E4,F4> . D4])])]]
         E,F)
```

and the following internalised function:

```
Δ[E0("LAMBDA)]                                                    (S4-521)
   = λS.λE.λF.λC .
       C("(E0("EXPR) ENV(E,F) NTH(1,S,F) NTH(2,S,F)),
         E,F)
```

where EXTEND is a function that extends environments according to the parameter matching protocols. If parameters were constrained to be single atoms (as, for example, in the λ-calculus), EXTEND would have the following simple definition:

```
EXTEND  : [[ ENVS × S × S ] → ENVS ]                              (S4-522)
   ≡ λE.λS1.λS2 .
        λA ∈ ATOMS if [A = S1] then S2 else E(A)
```

In fact we require a more complex EXTEND, because we support rail decomposition in the matching process; a correct version of EXTEND will be given below.

In order to support IMPRs and MACROs as well as EXPRs, however, we will adopt a different strategy from that exemplified in S4-620 and S4-621. The idea — one we will extend in 3-LISP — will be to have LAMBDA take three arguments, the first of which should designate a function that takes environments designators as well as parameter patterns and body expressions onto functions appropriately. Thus we will have the following simple definition:

```
Σ[E0("LAMBDA)]                                                    (S4-523)
   = λE.λF.λC .
       C("(IMPR E0 '[PARAM BODY] '(LAMBDA PARAM BODY)),
         [λS1.λE1.λF1 .
             Σ(NTH(1,S1,F1),E1,F1,
               [λ<S2,D2,E2,F2> . D2(E2,NTH(2,S1,F2),NTH(3,S1,F2))])]
         E,F)
```

and the following internalised function:

```
Δ[E0("LAMBDA)]                                                    (S4-524)
   = λS.λE.λF.λC .
       Σ(NTH(1,S,F),E,F,
         [λ<S1,D1,E1,F1> .
            Σ("(S1 HANDLE(ENV(E1,F1))
                   HANDLE(NTH(2,S,F1))
                   HANDLE(NTH(3,S,F1))),
              E1,F1,
              [λ<S2,D2,E2,F3> . C(S2,E2,F2)])])
```

Thus the import of a term like (LAMBDA EXPR [X] (+ X 1)) is carried by the significance of its first argument. Crucial, then, is the significance and internalisation of EXPR:

```
Σ[E0("EXPR)]                                                      (S4-525)
   = λE.λF.λC .
       C("(E0("EXPR) E0 '[ENV PARAM BODY] '(EXPR ENV PARAM BODY)),
         [λEc.λSp.λSb .
             [λ<S1,E1,F1> .
                 Σ(S1,E1,F1,
                   [λ<S2,D2,E2,F2> .
                       Σ(Sb,EXTEND(Ec,Sp,S2),F2,[λ<S3,D3,E3,F3> . D3])])]]
         E,F)

Δ[E0("EXPR)]                                                      (S4-526)
   = λS.λE.λF.λC .
       Σ(NTH(1,S,F),E,F,
         [λ<S1,D1,E1,F1> .
             Σ(NTH(2,S,F1),E1,F1,
               [λ<S2,D2,E2,F2> .
                   Σ(NTH(3,S,F2),E2,F2,
                     [λ<S3,D3,E3,F3> .
                         C("(E0("EXPR) S1 S2 S3),E3,F3)])])])
```

Note that S4-525 is recursive: the CAR of the closure returned as its result is itself; this was predicted at the end of chapter 3, and we assume the minimal circular solution, pictured in S3-200. Note as well that reductions in terms of EXPR are extraordinarily simple: they simply normalise each of the arguments, and return an application of identical form. Thus if X, Y, and Z are in normal form, (EXPR X Y Z) will normalise to (EXPR X Y Z).

It is the designation of E_0("*EXPR*) that is important and revealing. This term designates a function of three arguments: an enclosing environment E_c, a parameter structure S_p, and a body S_b. Since a LAMBDA term designates the application of this function designated by E_0("*EXPR*) to the environment in the context of use at the time of reduction in terms of LAMBDA, and to the parameter pattern and the body, this is as we expected. The EXPR function then designates a standard type of function that normalises its arguments, and that designates the designation of the body expression with respect to a context formed by the extension of the enclosing environment to include the binding of the parameter variables to the result of normalising the argument.

The significance of IMPR is of course similar, except that the arguments are not processed. Note however that the parameter pattern is matched against the *handle* of the arguments: thus the bindings remain in normal form, but a meta-level cross has transpired:

```
Σ[E0("IMPR)]                                                    (S4-527)
   = λE.λF.λC .
       C("(E0("EXPR) E0 '[ENV PARAM BODY] '(IMPR ENV PARAM BODY)),
         [λEc.λSp.λSb .
             [λ<S1,E1,F1> .
                 Σ(Sb,EXTEND(Ec,Sp,HANDLE(S1)),F1,[λ<S2,D2,E2,F2> . D2])]]
         E,F)

Δ[E0("IMPR)]                                                    (S4-528)
   = λS.λE.λF.λC .
       Σ(NTH(1,S,F),E,F,
         [λ<S1,D1,E1,F1> .
             Σ(NTH(2,S,F1),E1,F1,
               [λ<S2,D2,E2,F2> .
                   Σ(NTH(3,S,F2),E2,F2,
                     [λ<S3,D3,E3,F3> .
                         C("(E0("EXPR) S1 S2 S3),E3,F3)])])])
```

IMPR, of course, is an EXPR, as mentioned in the previous section.

On their own the three pairs of equations (S4-523 through S4-528) are not enough to discharge our obligations regarding closures: we need in addition to specify the internalised function signified by *non-primitive* closures. As we have characterised each primitive

procedure we have set out its internalised function, but of course in the general case the CAR of a redex will reduce not to a primitive closure but to one formed in terms of EXPR, IMPR (or MACRO once we have introduced that). In chapter 3 we gave a general characterisation of Δ for non-primitive closures in S3-137; what we need is a 2-LISP version of that equation.

Given the fact that all 2-LISP procedures are called with a single argument, the solution will be even simpler than that shown in S3-137. We have, in particular, the following:

$$\forall S_e, S_p, S_b \in \mathcal{S},\ E \in \mathit{ENVS} \qquad \text{(S4-529)}$$
$$[\![\, S_e = \mathrm{ENV}(E)\,] \supset$$
$$[\,\Delta\Gamma\text{"}(\langle \mathit{EXPR}\rangle\ \underline{S_e}\ \underline{\mathrm{HANDLE}(S_p)}\ \underline{\mathrm{HANDLE}(S_b)})\urcorner$$
$$= \lambda S_1.\lambda E_1.\lambda F_1.\lambda C_1$$
$$[\Sigma(S_1, E_1, F_1,$$
$$[\lambda\langle S_2, D_2, E_2, F_2\rangle\ .$$
$$\Sigma(S_b, E^*, F_2, [\lambda\langle S_3, D_3, E_3, F_3\rangle\ .\ C_1(S_3, E_3, F_3)])])]\,]\!]$$
$$\textit{where } E^* \textit{ is like } E \textit{ except extended by matching } S_2 \textit{ against } S_p.$$

The idea here is that S_e, S_p, and S_b are the environment, pattern, and body, respectively, of a *non-primitive* closure (S4-529 is intended to apply only to those closures whose internalisation is not otherwise specified). The internalised function signified by such a closure will be the function that, for any argument and context and continuation, first normalises the argument and calls the continuation with the result of normalising the body in an environment which is the closure environment extended as appropriate by binding the parameters in the pattern to the normalised argument.

The internalisation of non-primitive IMPR closures is similar but simpler, as expected:

$$\forall S_e, S_p, S_b \in \mathcal{S},\ E \in \mathit{ENVS} \qquad \text{(S4-530)}$$
$$[\![\, S_e = \mathrm{ENV}(E)\,] \supset$$
$$[\,\Delta\Gamma\text{"}(\langle \mathit{IMPR}\rangle\ \underline{S_e}\ \underline{\mathrm{HANDLE}(S_p)}\ \underline{\mathrm{HANDLE}(S_b)})\urcorner$$
$$= \lambda S_1.\lambda E_1.\lambda F_1.\lambda C_1$$
$$[\Sigma(S_b, E^*, F_1, [\lambda\langle S_2, D_2, E_2, F_2\rangle\ .\ C_1(S_2, E_2, F_2)])]\,]\!]$$
$$\textit{where } E^* \textit{ is like } E \textit{ except extended}$$
$$\textit{by matching } \mathrm{HANDLE}(S_1) \textit{ against } S_p.$$

The fact that the pattern in an IMPR are bound to *designators* of the argument expressions is reflected in the "HANDLE(S_1)" in the last line, plus the pattern matching extension adopted at the end of the last section.

An example will show how these equations entail that (LAMBDA EXPR [X] (+ X Y)) will designate (the extensionalisation of) an incrementation function if Y is bound to the

numeral 1 in the environment of reduction. In particular, we look at:

```
Σ("(LAMBDA EXPR [X] (+ X Y)),E1,F1,ID)                                    (S4-531)
```

where we assume that E_1("Y) = 1 and $E_1 = E_0$ otherwise. By the general significance of pairs (S4-38) we have

```
Σ("(LAMBDA EXPR [X] (+ X Y)),E1,F1,ID)                                    (S4-532)
  = Σ("LAMBDA,E1,F1,
      [λ<S1,D1,E1,F1> .
        [Δ(S1)]("[EXPR [X] (+ X Y)],E1,F1,
                [λ<S2,E2,F2> .
                   ID(S2,[D1("[EXPR [X] (+ X Y)]),E1,F1],E2,F2)])])
```

We can discard the unproductive ID, and discharge the initial binding of LAMBDA by using S4-523 that we just set forth:

```
  = ([λ<S1,D1,E1,F1> .                                                    (S4-533)
      [Δ(S1)]("[EXPR [X] (+ X Y)],E1,F1,
              [λ<S2,E2,F2> .
                 <S2,[D1("[EXPR [X] (+ X Y)],E1,F1)],E2,F2>])]
     <"(IMPR E0 [PARAM BODY] (LAMBDA PARAM BODY)),
      [λS1.λE1.λF1 .
         Σ(NTH(1,S1,F1),E1,F1,
           [λ<S2,D2,E2,F2> . D2(E2,NTH(2,S2,F2),NTH(3,S2,F2))])]
      E1,F1>)
```

We will choose to follow out the designational consequences first; when that is complete we will return and expand the internalised LAMBDA function. First, therefore, we reduce S4-533:

```
  = ([Δ("(IMPR E0 [PARAM BODY] (LAMBDA PARAM BODY)))]                     (S4-534)
     <"[EXPR [X] (+ X Y)],
      E1,
      F1,
      [λ<S2,E2,F2> .
          <S2,
           ([λS1.λE1.λF1 .
               Σ(NTH(1,S1,F1),E1,F1,
                 [λ<S2,D2,E2,F2> . D2(E2,NTH(2,S1,F2),NTH(3,S1,F2))])]
            <"[EXPR [X] (+ X Y)],E1,F1>)
           E2,F2>])
```

We can work now on the internal reductions here:

```
  = ([Δ("(IMPR E0 [PARAM BODY] (LAMBDA PARAM BODY)))]                     (S4-535)
     <"[EXPR [X] (+ X Y)],
      E1,
      F1,
      [λ<S2,E2,F2> .
          <S2,
           Σ(NTH(1,"[EXPR [X] (+ X Y)],F1),E1,F1,
             [λ<S2,D2,E2,F2> . D2(E2,
```

```
                                        NTH(2,"[EXPR [X] (+ X Y)],F2),
                                        NTH(3,"[EXPR [X] (+ X Y)],F2))]),
            E2,F2>])
```

Extracting the rail element on F_1:

```
= ([Δ("(IMPR E0 [PARAM BODY] (LAMBDA PARAM BODY)))]                    (S4-536)
   <"[EXPR [X] (+ X Y)],
    E1,
    F1,
    [λ<S2,E2,F2> .
        <S2,
         Σ("EXPR,E1,F1,
           [λ<S2,D2,E2,F2> . D2(E2,
                                        NTH(2,"[EXPR [X] (+ X Y)],F2),
                                        NTH(3,"[EXPR [X] (+ X Y)],F2))]),
         E2,F2>])
```

and applying S4-29 governing the general significance of atoms, in conjunction with S4-525:

```
= ([Δ("(IMPR E0 [PARAM BODY] (LAMBDA PARAM BODY)))]                    (S4-537)
   <"[EXPR [X] (+ X Y)],E1,F1,
    [λ<S2,E2,F2> .
        <S2,
         ([λ<S2,D2,E2,F2> . D2(E2,
                                NTH(2,"[EXPR [X] (+ X Y)],F2),
                                NTH(3,"[EXPR [X] (+ X Y)],F2))]
          <"(E0("EXPR) E0 [ENV PARAM BODY] (EXPR ENV PARAM BODY)),
           [λEc.λSp.λSb .
               [λ<S1,E1,F1> .
                   Σ(S1,E1,F1,
                     [λ<S2,D2,E2,F2> .
                         Σ(Sb,EXTEND(Ec,Sp,S2),F2,
                           [λ<S3,D3,E3,F3> . D3])])]]
           E1,F1>)
         E2,F2>])
```

This significance of E_0("EXPR) can be reduced:

```
= ([Δ("(IMPR E0 [PARAM BODY] (LAMBDA PARAM BODY)))]                    (S4-538)
   <"[EXPR [X] (+ X Y)],E1,F1,
    [λ<S2,E2,F2> .
        <S2,
         ([λEc.λSp.λSb .
               [λ<S1,E1,F1> .
                   Σ(S1,E1,F1,
                     [λ<S2,D2,E2,F2> .
                         Σ(Sb,EXTEND(Ec,Sp,S2),F2,
                           [λ<S3,D3,E3,F3> . D3])])]]
          <E1,NTH(2,"[EXPR [X] (+ X Y)],F1),
              NTH(3,"[EXPR [X] (+ X Y)],F1)>),
         E2,F2>])
```

And again, plus extracting the second and third rail elements out of F_1:

```
= ([Δ("(IMPR E0 [PARAM BODY] (LAMBDA PARAM BODY)))]                    (S4-639)
   <"[EXPR [X] (+ X Y)],E1,F1,
    [λ<S2,E2,F2> .
        <S2,
         [λ<S1,E1,F1> .
             Σ(S1,E1,F1,
               [λ<S2,D2,E2,F2> .
                    Σ("(+ X Y),EXTEND(E1,"[X],S2),F2,
                      [λ<S3,D3,E3,F3> . D3])])],
         E2,F2>])
```

This is as far as the designation will go: it is a function that accepts an argument (S_1) and a context (E_1 and F_1) and normalises its argument, and then designates the referent of `(+ X Y)` in the environment E_1, in which `Y` designates 1, extended with `X` designating whatever it designates in the calling context E_1.

We look then at the internalised LAMBDA function:

```
= ([λS.λE.λF.λC .                                                    (S4-640)
    Σ(NTH(1,S,F),E,F,
      [λ<S1,D1,E1,F1> .
        Σ("(S1 HANDLE(ENV(E1,F1))
               HANDLE(NTH(2,S,F1))
               HANDLE(NTH(3,S,F1))),E1,F1,
          [λ<S3,D3,E3,F3> . C(S3,E3,F3)])])]
   <"[EXPR [X] (+ X Y)],E1,F1,
    [λ<S2,E2,F2> .
      <S2,
       [λ<S1,E1,F1> .
          Σ(S1,E1,F1,
            [λ<S2,D2,E2,F2> .
                Σ("(+ X Y),EXTEND(E1,"[X],S2),F2,
                  [λ<S3,D3,E3,F3> . D3])])],
       E2,F2>])
```

and begin to reduce this (once again we do rail extractions immediately):

```
= Σ("EXPR,E1,F1,                                                     (S4-641)
     [λ<S1,D1,E1,F1> .
        Σ("(S1 HANDLE(ENV(E1,F1))
               HANDLE(NTH(2,"[EXPR [X] (+ X Y)],F1))
               HANDLE(NTH(3,"[EXPR [X] (+ X Y)],F1))),E1,F1,
          [λ<S3,D3,E3,F3> .
             ([λ<S2,E2,F2> .
                <S2,
                 [λ<S1,E1,F1> .
                    Σ(S1,E1,F1,
                      [λ<S2,D2,E2,F2> .
                          Σ("(+ X Y),EXTEND(E1,"[X],S2),F2,
                            [λ<S3,D3,E3,F3> . D3])])],
                 E2,F2>]
              <S3,E3,F3>)])])
```

Once again the significance of E_0("EXPR):

```
= ([λ<S1,D1,E1,F1> .
                                                                     (S4-642)
    Σ("(S1 HANDLE(ENV(E1,F1))
           HANDLE(NTH(2,"[EXPR [X] (+ X Y)],F1))
           HANDLE(NTH(3,"[EXPR [X] (+ X Y)],F1))),E1,F1,
      [λ<S3,D3,E3,F3> .
        ([λ<S2,E2,F2> .
          <S2,
           [λ<S1,E1,F1> .
              Σ(S1,E1,F1,
                [λ<S2,D2,E2,F2> .
                    Σ("(+ X Y),EXTEND(E1,"[X],S2),F2,
                      [λ<S3,D3,E3,F3> . D3])])],
           E2,F2>]
         <S3,E3,F3>)])]
```

```
<"(E0("EXPR) E0 [ENV PARAM BODY] (EXPR ENV PARAM BODY)),
  [λEc.λSp.λSb .
      [λ<S1,E1,F1> .
          Σ(S1,E1,F1,
            [λ<S2,D2,E2,F2> .
                Σ(Sb,EXTEND(Ec,Sp,S2),F2,[λ<S3,D3,E3,F3> . D3])])]]
  E1,F1>)
```

When we reduce this we will construct the appropriate pair with handles and so forth:

```
= Σ("((E0("EXPR) E0 [ENV PARAM BODY] (EXPR ENV PARAM BODY))          (S4-643)
     'E1 '[X] '(+ X Y))
   E1,F1,
   [λ<S3,D3,E3,F3> .
       ([λ<S2,E2,F2> .
           <S2,
            [λ<S1,E1,F1> .
                Σ(S1,E1,F1,
                  [λ<S2,D2,E2,F2> .
                      Σ("(+ X Y),EXTEND(E1,"[X],S2),F2,
                        [λ<S3,D3,E3,F3> . D3])])],
            E2,F2>]
        <S3,E3,F3>)])]
```

Now the first item here is of course the pair containing the primitive EXPR closure as its pair. From the general significance of pairs (S4-38) we have:

```
= Σ("(E0("EXPR) E0 [ENV PARAM BODY] (EXPR ENV PARAM BODY)            (S4-644)
   E1,F1,
   [λ<S1,D1,E1,F1>
      ([Δ(S1)]
       <F1²("((E0("EXPR) E0 [ENV PARAM BODY]
                            (EXPR ENV PARAM BODY))
                      'E1 '[X] '(+ X Y))),
        E1,F1,
   [λ<S2,E2,F2> .
       ([λ<S3,D3,E3,F3> .
           ([λ<S2,E2,F2> .
               <S2,
                [λ<S1,E1,F1> .
                    Σ(S1,E1,F1,
                      [λ<S2,D2,E2,F2> .
                          Σ("(+ X Y),EXTEND(E1,"[X],S2),F2,
                            [λ<S3,D3,E3,F3> . D3])])],
                E2,F2>]
            <S3,E3,F3>)]
        <S2,[D1(F1²("((E0("EXPR) E0 [ENV PARAM BODY]
                                    (EXPR ENV PARAM BODY))
                      'E1 '[X] '(+ X Y))),),E1,F1)],E2,F2>)])])
```

The primitive EXPR closure is in normal form and stable: thus we can simply abbreviate this expansion (otherwise it would cycle forever). However we need to know what the primitive closure *designates* (we will call this D• for the time being). We also do some F_1 field

extractions:

```
= ([λS.λE.λF.λC .                                                    (S4-646)
    Σ(NTH(1,S,F),E,F,
      [λ<S1,D1,E1,F1> .
          Σ(NTH(2,S,F1),E1,F1,
            [λ<S2,D2,E2,F2> .
                Σ(NTH(3,S,F2),E2,F2,
                  [λ<S3,D3,E3,F3> .
                      C("(E0("EXPR) S1 S2 S3),F,F)])])])]
   <"['E1 '[X] '(+ X Y)],
    E1,F1,
    [λ<S2,E2,F2> .
        ([λ<S3,D3,E3,F3> .
            ([λ<S2,E2,F2> .
                <S2,
                 [λ<S1,E1,F1> .
                     Σ(S1,E1,F1,
                       [λ<S2,D2,E2,F2> .
                           Σ("(+ X Y),EXTEND(E1,"[X],S2),F2,
                             [λ<S3,D3,E3,F3> . D3])])],
                 E2,F2>]
             <S3,E3,F3>)]
        <S2,[D*("(['E1 '[X] '(+ X Y)]),E1,F1)],E2,F2>)])])
```

Next EXPR normalises its arguments, but since they are all handles this is a straightforward (if messy) three steps (another α-reduction for perspicuity):

```
= Σ("'E1,E1,F1,                                                      (S4-646)
    [λ<S1,D1,E1,F1> .
        Σ(NTH(2,"['E1 '[X] '(+ X Y)],F1),E1,F1,
          [λ<S2,D2,E2,F2> .
              Σ(NTH(3,"['E1 '[X] '(+ X Y)],F2),E2,F2,
                [λ<S4,D4,E4,F4> .
                    ([λ<S2,E2,F2> .
                       ([λ<S3,D3,E3,F3> .
                          ([λ<S2,E2,F2> .
                             <S2,
                              [λ<S1,E1,F1> .
                                  Σ(S1,E1,F1,
                                    [λ<S2,D2,E2,F2> .
                                        Σ("(+ X Y),
                                          EXTEND(E1,"[X],S2),
                                          F2,
                                          [λ<S3,D3,E3,F3> . D3])])],
                             E2,F2>]
                          <S3,E3,F3>)]
                       <S2,[D*("(['E1 '[X] '(+ X Y)]),E1,F1)],
                        E2,F2>)]
                     ("(E0("EXPR) S1 S2 S4),E1,F1)])])])]

= Σ("'[X],E1,F1,                                                     (S4-647)
    [λ<S2,D2,E2,F2> .
        Σ(NTH(3,"['E1 '[X] '(+ X Y)],F2),E2,F2,
          [λ<S4,D4,E4,F4> .
              ([λ<S2,E2,F2> .
```

```
                    ([λ<S3,D3,E3,F3> .
                      ([λ<S2,E2,F2> .
                           <S2,
                            [λ<S1,E1,F1> .
                                Σ(S1,E1,F1,
                                  [λ<S2,D2,E2,F2> .
                                      Σ("(+ X Y),EXTEND(E1,"[X],S2),F2,
                                        [λ<S3,D3,E3,F3> . D3])])],
                            E2,F2>]
                       <S3,E3,F3>)]
                     <S2,[D*("([E1 '[X] '(+ X Y)]),E1,F1)],E2,F2>)]
                  <"(E0("EXPR) 'E1 S2 S4),E1,F1>)])])

= Σ("'(+ X Y),E1,F1,                                                    (S4-548)
    [λ<S4,D4,E4,F4> .
        ([λ<S2,E2,F2> .
           ([λ<S3,D3,E3,F3> .
              ([λ<S2,E2,F2> .
                   <S2,
                    [λ<S1,E1,F1> .
                        Σ(S1,E1,F1,
                          [λ<S2,D2,E2,F2> .
                              Σ("(+ X Y),EXTEND(E1,"[X],S2),F2,
                                [λ<S3,D3,E3,F3> . D3])])],
                    E2,F2>]
               <S3,E3,F3>)])]
            <S2,[D*("([E1 '[X] '(+ X Y)]),E1,F1)],E2,F2>)]
         <"(E0("EXPR) 'E1 '[X] S4),E1,F1>)])])

= ([λ<S2,E2,F2> .                                                       (S4-549)
     ([λ<S3,D3,E3,F3> .
        ([λ<S2,E2,F2> .
             <S2,
              [λ<S1,E1,F1> .
                  Σ(S1,E1,F1,
                    [λ<S2,D2,E2,F2> .
                        Σ("(+ X Y),EXTEND(E1,"[X],S2),F2,
                          [λ<S3,D3,E3,F3> . D3])])],
              E2,F2>]
         <S3,E3,F3>)])]
      <S2,[D*("([E1 '[X] '(+ X Y)]),E1,F1)],E2,F2>)]
   <"(E0("EXPR) 'E1 '[X] '(+ X Y)),E1,F1>)])])
```

We can now reduce this into the part of the signficance that is carrying the declarative import.

```
= ([λ<S3,D3,E3,F3> .                                                    (S4-550)
     ([λ<S2,E2,F2> .
          <S2,
           [λ<S1,E1,F1> .
               Σ(S1,E1,F1,
                 [λ<S2,D2,E2,F2> .
```

```
                                Σ("(+ X Y),EXTEND(E1,"[X],S2),F2,
                                  [λ<S3,D3,E3,F3> . D3])])],
                        E2,F2>]
                    <S3,E3,F3>))]
              <"(E0("EXPR) 'E1 '[X] '(+ X Y)),
               [D*("([E1 '[X] '(+ X Y)]),E1,F1)],
               E1,F1>)
```

Again:

```
= ([λ<S2,E2,F2> .                                                    (S4-661)
      <S2,
       [λ<S1,E1,F1> .
           Σ(S1,E1,F1,
             [λ<S2,D2,E2,F2> .
                 Σ("(+ X Y),EXTEND(E1,"[X],S2),F2,
                   [λ<S3,D3,E3,F3> . D3])])],
       E2,F2>]
   <"(E0("EXPR) 'E1 '[X] '(+ X Y)),E1,F1>)
```

And again:

```
= <"(E0("EXPR) 'E1 '[X] '(+ X Y)),                                   (S4-662)
   [λ<S1,E1,F1> .
       Σ(S1,E1,F1,
         [λ<S2,D2,E2,F2> .
             Σ("(+ X Y),EXTEND(E1,"[X],S2),F2,
               [λ<S3,D3,E3,F3> . D3])])],
   E1,F1>
```

We are then done (no more β-reductions apply). The full significance, then of (LAMBDA EXPR [X] Y) in E_1 is as expected. The *result* — the local procedural consequence — of this expression is a pair, the CAR of which is the primitive EXPR closure, reduced with three arguments: a *designator* of a structural encoding of E_1, and the parameter pattern and the body expression of the lambda form. This is just the closure we predicted. In order to know what this closure *designates*, however we look at the second element of the sequence. We see that it designates the following form:

```
[λ<S1,E1,F1> .                                                       (S4-663)
    Σ(S1,E1,F1,
      [λ<S2,D2,E2,F2> .
          Σ("(+ X Y),EXTEND(E1,"[X],S2),F2,
            [λ<S3,D3,E3,F3> . D3])])]
```

We recall from S4-168 that the definition of the extensionalising function is as follows:

```
EXT ≡ λG . [λS.λE.λF .
                                                                     (S4-664)
            Σ(S,E,F,
                [λ<S1,D1,E1,F1> . G(D1^1,D1^2, ... D1^k)])]
```

These are of course very similar in structure. The designation of (LAMBDA EXPR [X] (+ X Y)) can therefore be seen in the following light: it is a function of a structure and a context C_1 (i.e. E_2F_2), that maps that structure onto the *designation* of the expression (+ X Y) in a context C_2 which is like C_1 except that it is modified so that in it the variable X will be bound to the normalisation of its argument in C_1. This is also correct.

The term Σ("(+ X Y),EXTEND ...) could in turn be expanded, in conjunction with what know about E_1 — namely, that Y is bound to the numeral 1 — to prove that this is in fact the incrementation function. We will not do so here; we have merely shown how our characterisations do indeed carry the weight which we wanted them to. What we will do, in conclusion, is very simply show the significance of

```
((LAMBDA EXPR [X] (+ X Y)) 3)                                    (S4-666)
```

in an environment in which Y is bound to 1. A quick application of S4-38 yields:

```
Σ("((LAMBDA EXPR [X] (+ X Y)) 3),E1,F1,ID)                       (S4-666)
   = Σ("(LAMBDA EXPR [X] (+ X Y)),E1,F1,
       [λ<S1,D1,E1,F1> .
         [Δ(S1)]("[3],E1,F1,[λ<S2,E2,F2> . C(S2,D1("[3],E1,F1),E2,F2)])])
```

But of course we have just computed the first major part of this; therefore this reduces straight away to:

```
= ([Δ("(E0("EXPR) 'E1 '[X] '(+ X Y)))]                           (S4-667)
   <"[3],E1,F1,
    [λ<S2,E2,F2> .
        <S2,
         ([λ<S1,E1,F1> .
              Σ(S1,E1,F1,
                [λ<S2,D2,E2,F2> .
                     Σ("(+ X Y),
                       EXTEND(E1,"[X],S2),
                       F2,
                       [λ<S3,D3,E3,F3> . D3])])]
          <"[3],E1,F1>),
         E2,
         F2>])
```

Reducing first the inner application:

```
= ([Δ("(E0("EXPR) 'E1 '[X] '(+ X Y))})]                          (S4-668)
   <"[3],E1,F1,
    [λ<S2,E2,F2> .
        <S2,
         Σ("[3],E1,F1,
           [λ<S2,D2,E2,F2> .
```

```
                Σ("(+ X Y),
                  EXTEND(E1,"[X],S2),
                  F2,
                  [λ<S3,D3,E3,F3> . D3])]),
          E2,
          F2>])
```

We will do the inner (declarative) semantics first, in one step assuming that `[3]` self-normalises and designates <3> without side-effects:

```
= ([Δ("(E0("EXPR) 'E1 '[X] '(+ X Y)))]                                    (S4-559)
   <"[3],E1,F1,
    [λ<S2,E2,F2> .
        <S2,
         Σ("(+ X Y),
           EXTEND(E1,"[X],"[3]),
           F1,
           [λ<S3,D3,E3,F3> . D3]),
         E2,
         F2>])
```

Again we can assume from prior examples that `(+ X Y)` in an environment in which `X` is bound to `3` and `Y` to `1` (as `EXTEND(E1,"[X],"[3])` will of course ensure), will designate the number 4:

```
= ([Δ("(E0("EXPR) 'E1 '[X] '(+ X Y)))]                                    (S4-560)
   <"[3],E1,F1,[λ<S2,E2,F2> . <S2,4,E2,F2>]>)
```

We are now ready to apply the internalisation of general `EXPR` closures set out in `S4-529`:

```
= ([λS1.λE1.λF1.λC1 .                                                     (S4-561)
     [Σ(S1,E1,F1,
        [λ<S2,D2,E2,F2> .
           Σ("(+ X Y),E*,F2,[λ<S3,D3,E3,F3> . C1(S3,E3,F3)])])]]
   <"[3],E1,F1,[λ<S2,E2,F2> . <S2,4,E2,F2>]>)
    where E* is like E1 except extended by matching S2 against "[X]
```

A simple reduction:

```
= Σ("[3],E1,F1,                                                           (S4-562)
    [λ<S2,D2,E2,F2> .
       Σ("(+ X Y),E*,F2,
         [λ<S3,D3,E3,F3> .
            ([λ<S2,E2,F2> . <S2,4,E2,F2>]
             <S3,E3,F3>)])]
     where E* is like E1 except extended by matching S2 against "[X]
```

Once again we omit the simple derivation of the significance of `[3]`:

```
= Σ("(+ X Y),E*,F1,                                                       (S4-563)
    [λ<S3,D3,E3,F3> .
       ([λ<S2,E2,F2> . <S2,4,E2,F2>]
        <S3,E3,F3>)])
```

where E^* *is like* E_1 *except extended by matching* "[3] *against* "[X]

Note that we have updated our account of E^*. But again we assume that X is bound to 3 in E^*; thus we can again assume that (+ X Y) will normalise to 4 and designate 4:

$$= ([\lambda\langle S_2, E_2, F_2\rangle \ . \ \langle S_2, 4, E_2, F_2\rangle] \ \langle \text{"4}, E^*, F_1\rangle)]) \qquad \text{(S4-564)}$$

which finally reduces to our answer:

$$= \langle \text{"4}, 4, E^*, F_1\rangle \qquad \text{(S4-565)}$$

In other words the field remains unchanged, and the modified environment E^* is returned (but that is of no consequence since we would discard it above). The original expression thus designates four and returns the appropriate numeral. As expected.

4.c.v. Recursion

We have discussed the use of λ-terms as function designators without mentioning the subject of *recursion.* This approach was intentional, for it is important to ground the notion of a λ-term in the simpler case, but it is of course necessary to face the question of recursive definitions. We have used DEFINE with recursive λ-terms in many prior examples, but it should be clear that none of the discussion in the preceding section explains how they might be made to work.

As usual the first task is to make clear what we mean by the term. In recent years the notion of a *recursive* procedure has been increasingly contrasted with that of an *iterative* procedure, based on the intuition that certain functions that were traditionally considered recursive are in some deeper sense not really recursive at all — they don't appear to have the fundamental properties (such as requiring memory in proportion to depth of call) characteristic of the "paradigmatic" recursive functions like factorial. On the other hand there is a sense that any definition using its own name within it is recursive. Finally, there are a variety of mathematical concepts: of a "recursive" function, deriving from the notion of composition of a certain set of functional ingredients; of a recursive set — a set with a decidable characteristic function; and so forth.

What is of concern here is what the predicate "recursive" is being applied to. There are in particular three ways in which we may use the term, of increasing semantic depth: as applying to *signs,* to *intensions,* and to *extensions.* The original and most accessible notion of recursion is as a predicate on *function designators* — on *signs,* in other words: a *definition* of a function is recursive just in case a term designating the whole function is used as an ingredient term within the definition. This is the sense in which LISP is said to support recursion and FORTRAN not; it is also the kind of recursion we mean when we say that a semantic domain is *recursively* defined by an equation such as $D \simeq [\, D \times D \,]$. However it is of course a consequence of this definition that nothing of interest can be said about the class of functions designated by ***recursive definitions***, since for *any* function designator F we can construct the following designator F' that it recursive, on this syntactic account, but that designates the same function:

```
F' ≡ λX . if [1 = 0] then F'(X+1) else F(X)                    (S4-670)
```

Furthermore, there is no way mechanically to exclude such definitions — those involving only *gratuitous* recursive uses of the function's name in the body of the definition — since it is not in general decidable whether a recursive use of that name plays an essential role.

This, then, is the most "hyper-intensional" use of the predicate. The non-semantic character of this use of the word will emerge strongly below, where we show how Church's Y operator enables any recursive definition on our account to be converted into a non-recursive definition, in virtue of the use of an explicit non (syntactically) recursive designator of the fixed-point function. In other words, not only can all non-recursive definitions be rendered recursive by the technique illustrated in S4-670; far more importantly, all *recursive* definitions can be rendered *non-recursive* in conjunction with the fix-point function. In sum, syntactic recursion, as Church and others have showed, can always be discharged. Nonetheless, it is with the support of recursive definitions — recursive *lambda terms* — that we are concerned in this section. Our formalism — the 2-LISP architecture we have already adopted — is fully adequate to support arbitrary ***recursive functions***, general as well as elementary, in the technical sense,[6] no matter how they are designated.

Midway between the hyper-intensional notion of a recursive *definition* and the extensional notion of a recursive *function* is the notion of what we will call a ***recursive procedure*** — a use of the term "recursive" over *functions in intension.* The "iterative-recursive" distinction of computer science trades on this intensional use of the term; it is worth mentioning because it will matter in our characterisation of the meta-circular and reflective processors we will encounter in later sessions. The intuition is exemplified by the following two definitions of factorial: though both are *syntactically recursive*, and although they are *extensionally identical* (they both designate the factorial function), there is a point of view — an *intensional* point of view, again — by which they are different. The processing of the first, in a depth-first "recursive" control regime (that's of course yet a fourth notion of "recursive", having more to do with compositionality, although the structure of the natural designators of such a processing function are typically recursive in form — thus it is not an *unrelated* notion), requires a finite but indefinite amount of memory, whereas processing the the third, a sub-procedure to the second, requires a fixed

(and small) amount of storage, independent of the magnitude of the argument (although of course the representation of the answer does require storage that grows with the depth of the processing):

```
(DEFINE FACTORIAL1                                                    (S4-571)
   (LAMBDA EXPR [N]
      (IF (= N 0)
          1
          (* N (FACTORIAL (- N 1))))))

(DEFINE FACTORIAL2                                                    (S4-572)
   (LAMBDA EXPR [N] (FACTORIAL2-HELPER 0 1 N)))

(DEFINE FACTORIAL2-HELPER                                             (S4-573)
   (LAMBDA EXPR [COUNT ANSWER N]
      (IF (= COUNT N)
          ANSWER
          (FACTORIAL2-HELPER (+ COUNT 1) (* ANSWER (+ COUNT 1)) N))))
```

In the processors we will define for 2-LISP and 3-LISP it will turn out that the essentially "iterative" (non-increasing storage) nature of FACTORIAL$_2$-HELPER is embodied in its processing, since the "embedding" of the continuation structure (as was pointed out by Steele and Sussman) is engendered by the processing of arguments, not by the reduction of procedures. The intensional distinction, in other words, matters to us, and is adequately treated in our meta-theoretic accounts. Furthermore, this fact will play a crucial role in our ability to claim that the entire state of processing of a 3-LISP procedure is contained at a given reflective level, since our defense will involve a recognition of the fact that all embedded calls to the processor function — the "recursion" mentioned above that characterise the basic LISP control regime — are "tail-recursive" in the sense of FACTORIAL$_2$-HELPER, thus requiring no maintence of state on the part of *its* processor. But these are all matters for a later section. Our present concern is merely with what we will call ***syntactic recursion*** in LAMBDA terms.

Note that, in spite of an informal sense that syntactic recursion involves some sort of *self-reference*, the kind of recursion we are concerned with here involves the embedded *use* of a name for the procedure, not a *mention* of that name. Syntactic recursion, in other words, is not *self-reference* of the sort that will permeate our discussions of reflection in the next chapter. In order to see this clearly, consider again the recursive definition of FACTORIAL in S4-571 above. The LAMBDA term is a *sign*, with some intensional content, that *designates* the factorial function. The embedded use of the name FACTORIAL is intended

also to designate that function. However neither the LAMBDA term, nor the FACTORIAL term, nor any of the other constituents, designate the LAMBDA term or the FACTORIAL term or any of the other constituents. Nor does any part of the factorial function *qua* abstract mathematical function contain any designators at all (both the domain and range of the function are *numbers*, not *signs*). Though it is by no means easy to make the concept of self-reference precise, the notion would seem at heart to have something to do with a syntactic or linguistic object that either was (or was part of) *its own designation.* No amount of *circularity* or *recursion* trades on any such *mentioning* of a designator by that designator. Syntactic recursion, in other words, to the extent there is anything "self"-ish about it, involves a kind of ***self-use***, rather than true *self-reference.*

There is a received understanding in the community that a proper or adequate implementation of recursive definitions requires in a deep sense some kind of *circularity* on the part of the implementing mechanism. We will ultimately show that this is true, but that it is not *obviously* true was shown by Church (as part of a proof of the universal power of the λ-calculus) in his demonstration of the *paradoxical combinator* or *Y operator*, an apparently non-circular and non-recursive (in the syntactic sense) term that designates what has come to be known as the ***fixed point function***. The pure λ-calculus form of the Y operator is as follows:

```
λF . ((λX . F(X(X)))                                          (S4-574)
      (λX . F(X(X))))
```

This would be used as follows. Suppose we had the following incomplete definition of FACTORIAL — incomplete because the the term FACT is unbound (this is expressed in our eclectic meta-language, not the pure λ-calculus, since we are assuming numerical arguments and other primitive functions, but the idea is clear):

```
λN . [if [N = 0] then 1 else [N * FACT(N-1)]]                 (S4-575)
```

Then the insight — the fundamental content of the notion of a fixed point — is that this would designate the factorial function if FACT were bound to the factorial function. If, more particularly, we had the expression:

```
H ≡ λFACT . [λN . [if [N = 0] then 1 else [N * FACT(N-1)]]]   (S4-576)
```

then if H were applied to the actual factorial function, then the value would be that factorial function. If, for example, we *knew* that G designated the factorial function, then H(G)

would designate it as well.

To this intuition is added the realisation that S4-675 contains everything necessary to specify the factorial function. It is then standard theory to show that the Y operator correctly embodies this intuition. The result is that Y(H) designates the factorial function. To continue with our mixed meta-language, one would show that

```
((Y(H)) 6)                                                    (S4-677)
```

designated the number 720, and so forth.

This is all elementary. We have reviewed it because we can, if we wish, absorb this behaviour almost directly into 2-LISP. This is particularly useful because we will be able to define a declaratively and behaviourally adequate procedure that will enable us to handle all kinds of recursion (single and mutual recursion, top-level definitions of recursive procedures, and so forth) — all without requiring us to define any new primitive mechanisms to extend those we have already adopted. We will not ultimately employ the procedure that results, since it involves some unavoidable conceptual ineffeciencies, but we will base the (non-primitive) procedure we do finally select on our translation of Church's function.

Before setting out on this project, we should admit straight away that the techniques by which recursive definitions are supported in standard LISPs must be rejected. By and large definitions are allowed only at the so-called "top-level" (one cannot use DEFINE embedded within a program); the bindings that result are established globally, in special function cells. Since these standard LISPs are dynamically scoped, any recursive use of the name of a procedure within that procedure's body will of necessity find the binding already established, *when the procedure is used*, since the binding will have been constructed at an earlier period of time, and there is only a single space of procedure definitions. We cannot accept this protocol for a variety of reasons:

1. We want to be able to embed definitions, particularly within the scope of bindings (in forms such as (LET [[X 1]] (DEFINE F (LAMBDA ... X ...))), for example). Useful in general, this kind of practice is particularly natural in a statically-scoped dialect.
2. We do not store "functional" properties differently from standard "values": the binding of procedure names must therefore use the same mechanisms as those used to support general binding. It would be awkward if one could both LAMBDA bind and SET variables in general, but only SET could be used to bind

names to recursive definitions.

3. It is not possible to use the traditional mechanism to implement mutually-recursive sub-procedures visible only within a specific context. In LISP 1.5 a separate primitive called LABEL was provided for this case. There is no defensible reason to need an extra primitive.
4. The success of the recursive nature of the definition arises rather accidentally out of global properties of the system, which is inelegant.

Furthermore, as the analysis of the next pages demonstrates, the acceptance of the standard techniques overlooks some distinctions of considerable importance, that a close look at the fixed point function will bring into explicit focus. As well as defining LABEL and DEFINE as simple non-primitive functions, we will be able to provide such facilities as protecting bindings in a closure from the impact of subsequent DEFINEs, all without resorting to special purpose mechanisms.

Some of our complaints are of course handled by Sussman and Steele's SCHEME, but even that dialect does not support embedded definitions, in spite of its static scoping (SCHEME has LABELS, but does not support (LET ((A 1)) (DEFINE ...))). In sum, procedure definition has to date received rather *ad hoc* treatment, something we should attempt to repair.

We turn then to Church's Y operator. It is a straightforward function: it is of course of indefinite order, since it applies to functions, but since 2-LISP is an untyped higher-order formalism, no trouble will arise in using such a function in our dialect. Suppose, for example, we define the following initial 2-LISP version (a certain circularity in our pedagogical style should be admitted: we are using DEFINE to define functions that we will ultimately use in order to explain what DEFINE does, but so be it) — this is merely a syntactic transformation into 2-LISP of S4-574:

```
(DEFINE Y1                                                    (S4-578)
   (LAMBDA EXPR [FUN]
      ((LAMBDA EXPR [X] (FUN (X X)))
       (LAMBDA EXPR [X] (FUN (X X))))))
```

This can be more perspicuously written as follows (although the Y operator has never been the most pedagogically accessible of functions):

```
(DEFINE Y1                                                    (S4-579)
   (LAMBDA EXPR [FUN]
      (LET [[X (LAMBDA EXPR [X] (FUN (X X)))]]
         (FUN (X X)))))
```

The idea would be that if we defined a version of factorial as follows:

```
(DEFINE H                                                    (S4-680)
   (LAMBDA EXPR [FACT]
      (LAMBDA EXPR [N]
         (IF (= N 0)
             1
             (* N (FACT (- N 1)))))))
```

then we *should* have:

```
((Y1 H) 1)     ⇒     1          ; This is desired, but not      (S4-681)
((Y1 H) 6)     ⇒   720          ; actual, behaviour.
```

and so forth.

Although Y_1 is declaratively correct, any use of it will fail, for procedural reasons. The problem is that although $(Y_1\ H)$ can be shown to *designate* the factorial function, the processing of $(Y_1\ H)$ would never *return*. It would engender an infinite computation before it ever returned a procedure to reduce with a numeric argument. This trouble is apparent from a brief examination of how Y_1 works. Y_1 gives to the function H a procedure that embeds not only another copy of H, but a copy of the application of the Y operator to H, so that it thereby engenders an infinite embedded tree of procedure definitions. This is all very well *declaratively*, since that is really what the recursive use of the name means, but it is less acceptable *procedurally*, since we do not wish actually to generate this infinite tree of procedure expressions, which is what Y_1 does.

In the λ-calculus, as we have noted before in conjunction with the conditional, the reduction protocols are *normal order*, rather than *applicative order*. Y_1 would work properly in a normal-order system; to be LISP, however, we will require an adequate applicative order variant.

This problem, however, is easily repaired. In section 4.c.i we discussed the fact that wrapping a designating expression in a LAMBDA term and then reducing a corresponding redex at a different time is a standard way of deferring the processing of intensions. Using this technique, it is straightforward to define a modified version of Y_1, to be called Y_2, that defers processing of each embedded application of itself until the arguments have been given to the recursive procedure. Thus Y_2 alternately reduces one argument set, then one self-application, then one argument set, and so on, back and forth. Note the use of a single atom ARGS for a pattern, enabling Y_2 to be used for procedures of any number of arguments:

```
(DEFINE Y2                                                    (S4-582)
   (LAMBDA EXPR [FUN]
      ((LAMBDA EXPR [X] (FUN (X X)))
       (LAMBDA EXPR [X]
          (LAMBDA EXPR ARGS
             ((FUN (X X)) . ARGS)))))
```

or, once again to use LET:

```
(DEFINE Y2                                                    (S4-583)
   (LAMBDA EXPR [FUN]
      (LET [[X (LAMBDA EXPR [X]
                  (LAMBDA EXPR ARGS ((FUN (X X)) . ARGS)))]]
         (LAMBDA EXPR [X] (FUN (X X))))))
```

Y_2 is acceptable, both declaratively and procedurally, for single recursive procedures of any number of arguments, providing, of course, that they are EXPRs (we will discuss recursive IMPR definitions presently). We have, for example, the following actual 2-LISP behaviour (we avoid DEFINE here merely to illustrate how Y_2 frees us from any need to have DEFINE perform any sort of magic):

```
> (SET G                                                      (S4-684)
       (Y2 (LAMBDA EXPR [FACT]
              (LAMBDA EXPR [N]
                 (IF (= N 0)
                     1
                     (* N (FACT (- N 1))))))))
> G
> (G 0)
> 1
> (G 6)
> 720
> (G (G 4))
> 6204484017332394393600000
```

This illustration brings up a point we will consider at considerable length below: what it is to give to a surrounding context a *name* for a recursive procedure. In the example we used the name "G" — different from the name FACT used internally. It is of course normal, and simple, to have the name in the environment and the name within the procedure definition be the same, but our approach has shown how these are at heart two different issues. The name, in any particular context, by which a procedure is known is a matter *of that context*, whereas the name used *within a recursive LAMBDA term* to refer to itself is a matter *of the LAMBDA intension*. As we have said before, with a better theory of intension we might escape having to retain the internal name at all (for example, although this cannot be done, because of decidability considerations, one can imagine replacing all recursive instances of

the name replaced with actual references to the resultant closure). Thus in our discussion of environments, which arise in connection with a suitable definition for DEFINE, we must not be led into confusion about the recursive aspects of the LAMBDA term. As Church has shown, and we have adapted to 2-LISP's circumstances, the latter concern can be treated adequately and independently of the former.

Before turning to those issues, however, there are a variety of concerns with Y_2 that we should attend to, if we are going to base subsequent definitions of other variants on its external behaviour. First, as given it is unclear how we might support *mutually recursive* definitions. Algorithmic procedures do exist whereby two or more mutually recursive definitions can be "unwound" into a single recursive one, but it is convenient nonetheless to generalise the definition of Y to encompass more than one definition. It is convenient to have an example. Though there are familiar cases of mutually recursive definitions (the EVAL and APPLY of 1-LISP are a familiar pair), they tend to be rather complex; we will therefore consider the following two rather curious mutually-defined functions: it can be seen on inspection that (G1 A B) designates either A or B, depending on whether the product of A and B is odd or even, respectively:

```
(DEFINE G1                                                    (S4-585)
   (LAMBDA EXPR [A B]
      ((G2 A B) (+ A B) A)))

(DEFINE G2
   (LAMBDA EXPR [A B]
      (IF (EVEN (* A B))
          -
          G1)))
```

Thus we have, for example:

```
(G1 3 4)     ⇒     4                                          (S4-586)
(G1 4 3)     ⇒     3
(G1 5 7)     ⇒     5
```

It is clear that any fixed-point abstraction over mutually recursive defintions will have to bind formal parameters to *all* of the elements of the mutually recursive set, since applications in terms of any of them may appear within the scope of each definition. Thus we will have to treat the following two fixed point expressions:

```
H1 ≡ (LAMBDA EXPR [G1 G2]                                     (S4-587)
       (LAMBDA EXPR [A B]
          ((G2 A B) (+ A B) A)))
```

```
H2 ≡ (LAMBDA EXPR [G1 G2]                                        (S4-688)
       (LAMBDA EXPR [A B]
         (IF (EVEN (* A B))
             -
             G1)))
```

An appropriate 2-function variant on Y_2, called YY_2, is defined in S4-690, below. The term (YY_2 H1 H2) designates a two element sequence of the two functions in question; thus we would expect the following behaviour:

```
((NTH 1 (YY2 H1 H2)) 3 4)      ⇒    4                            (S4-689)
((NTH 1 (YY2 H1 H2)) 4 3)      ⇒    3
((NTH 1 (YY2 H1 H2)) 6 7)      ⇒    6
```

The definition of YY_2 is this:

```
(DEFINE YY2                                                      (S4-690)
   (LAMBDA EXPR [FUN1 FUN2]
      ((LAMBDA EXPR [X1 X2]
         [(FUN1 (X1 X1 X2) (X2 X1 X2))
          (FUN2 (X1 X1 X2) (X2 X1 X2))])
       (LAMBDA EXPR [X1 X2]
          (LAMBDA EXPR ARGS
             ((FUN1 (X1 X1 X2) (X2 X1 X2)) . ARGS)))
       (LAMBDA EXPR [X1 X2]
          (LAMBDA EXPR ARGS
             ((FUN2 (X1 X1 X2) (X2 X1 X2)) . ARGS))))))
```

Once again we present a LET version for those who find this clearer:

```
(DEFINE YY2                                                      (S4-691)
  (LAMBDA EXPR [FUN1 FUN2]
     (LET [[X1 (LAMBDA EXPR [X1 X2]
                 (LAMBDA EXPR ARGS
                    ((FUN1 (X1 X1 X2) (X2 X1 X2)) . ARGS)))]
           [X2 (LAMBDA EXPR [X1 X2]
                 (LAMBDA EXPR ARGS
                    ((FUN2 (X1 X1 X2) (X2 X1 X2)) . ARGS)))]]
       (LAMBDA EXPR [X1 X2]
          [(FUN1 (X1 X1 X2) (X2 X1 X2))
           (FUN2 (X1 X1 X2) (X2 X1 X2))]))))
```

This indeed supports the behaviour indicated in S4-689, both declaratively and procedurally.

It is of course necessary to generalise once more. Y_2* will accept an arbitrary number of mutually recursive definitions, and will designate a sequence of the functions they designate. It therefore follows that the normalisation of

```
(Y2* F1 F2 ... Fk)                                               (S4-692)
```

will return a normal-form rail of the appropriately defined closures of the functions in question. Y_2* may be defined as follows:

```
(DEFINE Y2*                                                          (S4-593)
   (LAMBDA EXPR FUNS
      ((LAMBDA EXPR [RECS]
         (MAP (LAMBDA EXPR [FUN]
                 (FUN . (MAP (LAMBDA EXPR [REC] (REC . RECS))
                             RECS)))
              FUNS))
       (MAP (LAMBDA EXPR [FUN]
              (LAMBDA EXPR RECS
                 (LAMBDA EXPR ARGS
                    ((FUN . (MAP (LAMBDA EXPR [REC] (REC . RECS))
                                 RECS))
                     . ARGS))))
            FUNS))))
```

Again a LET version:

```
(DEFINE Y2*                                                          (S4-594)
   (LAMBDA EXPR FUNS
      (LET [[RECS
             (MAP (LAMBDA EXPR [FUN]
                    (LAMBDA EXPR RECS
                       (LAMBDA EXPR ARGS
                          ((FUN . (MAP (LAMBDA EXPR [REC] (REC . RECS))
                                       RECS))
                           . ARGS))))
                  FUNS)]]
        (MAP (LAMBDA EXPR [FUN]
               (FUN . (MAP (LAMBDA EXPR [REC] (REC . RECS))
                           RECS)))
             FUNS))))
```

Note the substantial use of non-rail argument forms, facilitating the fact that Y* can be used with an arbitrary number of mutually recursive definitions.

Such a definition, of course, though of theoretical interest, would in a practical setting never be used explicitly. What is striking is that we can define the standard LISP notion of LABELS directly in terms of Y_2*. Assume, in particular, that LABELS is a macro that expands expressions of the form:

```
(LABELS [[<L1> (LAMBDA ... <E1> ... )]                               (S4-595)
         [<L2> (LAMBDA ... <E2> ... )]
         ...
         [<Lk> (LAMBDA ... <Ek> ... )]]
    <BODY>)
```

into expressions as follows:

```
(LET [[[<L1> <L2> ... <Lk>]                                          (S4-596)
       (Y2* (LAMBDA EXPR [<L1> <L2> ... <Lk>] (LAMBDA ... <E1> ...))
            (LAMBDA EXPR [<L1> <L2> ... <Lk>] (LAMBDA ... <E2> ...))
              ...
            (LAMBDA EXPR [<L1> <L2> ... <Lk>] (LAMBDA ... <Ek> ...)))]]
   <BODY>)
```

This is then the standard LISP 1.6-style LABELS, defined as a user function. We would have, for example:

```
(LABELS [[G1 (LAMBDA EXPR [X Y] ((G2 X Y) (+ X Y) X))]               (S4-597)
         [G2 (LAMBDA EXPR [X Y]
                (IF (EVEN (* X Y)) - G1))]]
        (= (G1 2 3) (G1 3 5)))                          ⇒  $T
```

Note as well that the definition of the LABELS macro makes explicit what we mentioned earlier: it is standard, but not necessary, to have the name *within* the intensional expression and *external* to the intensional expression be the same.

Given this definition of Y, it is straightforward to define a first version of DEFINE in its terms. In particular, we can assume that expressions of the form:

```
(DEFINE <LABEL> <PROCEDURE>)                                         (S4-598)
```

are macro abbreviations of

```
(SET <LABEL> (Y2 (LAMBDA EXPR [<LABEL>] <PROCEDURE>)))               (S4-599)
```

In addition, to facilitate mutually recursive "top-level" definitions, it is straightforward to assume equivalently that expressions of the form

```
(DEFINE* <LABEL1> <PROCEDURE1>                                       (S4-600)
         <LABEL2> <PROCEDURE2>
         ...
         <LABELk> <PROCEDUREk>)
```

are abbreviations for

```
(MAP SET [<LABEL1> <LABEL2> ... <LABELk>]                            (S4-601)
     (Y2* <PROCEDURE1> <PROCEDURE2> ... <PROCEDUREk>))
```

Since DEFINE* is a pure extension of DEFINE — i.e., since DEFINE* and DEFINE are equivalent in effect when given a single argument — we might just as well assume the entire set of behaviours under the single name DEFINE.

One issue this agreement raises is this: we have assumed throughout that DEFINE need not be used with explicit LAMBDA terms. In particular, we have assumed we could use

it for simpler purposes, such as simply to give to one name the significance of another. For example, suppose we wish to give to the atom EQ the approximate meaning it has in 1-LISP, which in 2-LISP we have initially bound to the atom "=". The following would suffice:

```
(DEFINE EQ =)                                                    (S4-602)
```

Now it follows, from our present analysis, that the following would be equivalent in effect:

```
(SET EQ =)                                                       (S4-603)
```

Thus even if our new definition of DEFINE did not support S4-602, no power would be lost (if, in other words, S4-602 fails on our new definition, we could always simply use S4-603). However it is reassuring to recognise that S4-602 would simplify, under the expansion assumed in S4-599, to (the multiple argument Y_2* makes this immaterially more complex):

```
(SET EQ (Y2 (LAMBDA EXPR [EQ] =)))                               (S4-604)
```

which is equivalent, and thus in one sense correct, even though it employs complexity unnecessary to the circumstance. In particular, the atom = normalises to the primitive equality closure:

```
=                          ⇒    (<EXPR> E0 '[X Y] '(= X Y))      (S4-605)
```

However we also have:

```
(Y2 (LAMBDA EXPR [EQ] =))  ⇒    (<EXPR> E0 '[X Y] '(= X Y))      (S4-606)
```

The explict use of Y_2, in other words, has no discernible effect. The reason is simple: Y_2 in S4-604 causes the atom EQ to be bound to the fixed point defined over =. But = is bound to a normal-form closure; thus when it is normalised in this extended environment, the extended environment is thrown away (normal-forms are environment-independent). The = closure was closed in E_0 long before the reduction of Y_2 took place.

There would seem no disadvantage in using DEFINE in all cases, in other words, and this is how we will proceed. It should be admitted, however, for completeness, that there is *one* minor, but observable, difference in their behaviours. If EQ was already defined, the following code would have no visible effect:

```
(SET EQ EQ)                                                      (S4-607)
```

In this one case, however (where the second argument *uses* the term being set), the following is different in consequence:

```
(DEFINE EQ EQ)                                                    (S4=608)
```

In particular, S4-608 will define a viciously circular procedure, since it expands to

```
(SET EQ (Y2 (LAMBDA EXPR [EQ] EQ)))                               (S4=609)
```

which is content-free. Thus we should employ DEFINE with this one proviso, although it of course is hardly a seriously limitation (furthermore the explicit use of the simpler SET is always possible).

We said earlier that we would discuss at a later point the issue of how the names of recursively-defined procedures could be made available to a wider context; the present suggestions show how that question has reduced to one of making *any* structure or binding public. We have demonstrated, in particular, how to reduce questions of definitions and recursion (both single and multiple) to questions of setting variables — a subject on its own which we will take up explicitly in section 4.c.vi. What we intend to do in this particular section is to explore to its limits the question of constructing *function designators*; the issue of providing generally available names is separate.

There are, however, consequences of our approach to naming that emerge from our analysis of names. In particular, it is clearly possible to use DEFINE at other than the top level, thus embedding a potentially more complex context within the intension of the function defined. We have, for example:

```
(LET [[A 1]]                                                      (S4-610)
   (DEFINE INCREMENT (LAMBDA EXPR [N] (+ N A))))
```

where the resultant INCREMENT procedure will add the number 1 to its argument, independent of the binding of A in that argument's context. For example:

```
(LET [[A 3]] (INCREMENT A))      ⇒    4                           (S4-611)
```

Such an ability, perhaps surprisingly, is not available in any standard LISPs or in SCHEME; definitions being thought in some way to be remarkable — a view we are trying to dismantle.

Furthermore, equivalent in effect to S4-610 is the following:

```
(DEFINE INCREMENT                                                 (S4-612)
   (LET [[A 1]]
      (LAMBDA EXPR [N] (+ N A))))
```

This gives the procedure INCREMENT something like an OWN variable (this is because the binding of A does not occur within the *schematic scope* of the LAMBDA; hence there is an *instance* of A for the procedure as a whole, not a *schematic instance*, as there is for N — or to put it another way, there is one A for all instances of INCREMENT, but one instance of N per instance of INCREMENT). We could for example define a procedure that printed out how often it had been called:

```
(DEFINE COUNTER                                                    (S4-613)
   (LET [[COUNT 0]]
      (LAMBDA EXPR []
         (BLOCK (SET COUNT (INCREMENT COUNT))
                (PRINT COUNT)))))
```

yielding the following behaviour:

```
> (COUNTER) 1                                                      (S4-614)
> $T
> (IF (COUNTER) (COUNTER) (COUNTER)) 2 3
> $T
```

We have still some distance to go: we have not yet, for example, discussed intensional procedures (sadly, Y_2 will not work for recursive IMPRS). But before turning to that, we should make a comment: although we will not ultimately adopt the definition of DEFINE given in S4-599, because of the conceptual inefficiency of Y_2, it is useful to have shown the kind of behaviour it engenders, as illustrated in the last few examples. They provide an indication of the effect that any candidate treatment of recursion should honour. In other words, Y_2 will be a behavioural guide as we search for more effective variants on Y.

The problems with Y_2 have in part to do with efficiency — not *implementational* efficiency, but a deeper kind of conceptual inefficiency. This is best revealed by looking in more depth at how both Y_1 and Y_2 work, especially in light of the intuition that recursive definitions have in some way to do with circularity. On the face of it we apparently constructed a successful fixed point function defined as a pure tree, which reduced with other definitions that were pure trees, in such a way that the required recursive procedures emerged, without either Y or its arguments involving any circular or recursive use of self-names. We have shown, in other words, that we can eliminate syntactic recursion, without, apparently, introducing any structural circularity to do so (it is trivial to remove it *using* structural circularity, *if one knows where to perform the surgery*, as the example in chapter 2 illustrated). Thus it might seem that recursive definitions do not require circular structures, intuition not withstanding.

But intuition does in fact stand. The Y operator does construct circular structure, albeit of a particular sort. More specifically, an examination of the definition of Y_1 reveal that the Y operator works by constructing essentially indistinguishable copies of itself applied to the function in question, and at each application redoing this copying at infinitum. What is true about these structures is that they are *type-equivalent*, according to our definition of section 4.b.ii. The full (infinite) normalisation of Y_1 reduced with arguments, in other words, includes within it an infinite number of *type-equivalent* copies of itself.

We will say, therefore, that a closure returned by (Y$_1$ F) is ***type-circular***. Y_2 differs from Y_1 in that the closures it yields defer the production of this infinite type-circular structure so that one more embedded level of it is produced each time the closure is applied (strictly, each time the recursive self-name is used). The closures produced by applying Y_2, therefore, we will call ***type-circular-deferred***. In the λ-calculus, efficiency is not an issue, since the extension — the functions designated by the λ-terms — are of prime importance. Similarly, since there is no intensionality or side-effects, type-equivalent and token-equivalent (i.e., equal) structures are immaterially different. It is of no consequence in the λ-calculus, in other words, whether a structure is type-circular or token-circular (or, more carefully, it would be of no consequence: in fact token-circular structures cannot be constructed), since no behaviour hinges on the token identity of any given expression.

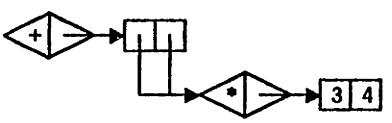
Given this analysis, it is natural to ask whether it is not possible to define an extensionally equivalent variant of the Y operator that traffics in ***token-circular*** closures: closures X such that [X ∈ ACCESSIBLE*(X)].

This understanding leads to a variety of suggestions, that we will explore in turn. First, it is instructive to imagine modifying the definition of Y itself, so that the *type-equivalent* structures within its definition are *in fact identical* — making, in other words, the definition of Y use identity in place of type-equivalence. To show this, we of course encounter a pedagogical difficulty: the resultant structures cannot be lexically notated. However since they will not be complex, we will adopt a simple extension of the standard notation, as follows. We will assume that any atom X followed immediately (no spaces intervening) by a colon, followed by a pair or rail, will define what we will call a ***notational label*** for that pair. Similarly, any occurrence of that label *immediately preceded* by a colon will stand in place of an occurrence of the pair following the place the label was defined. We will ignore scoping issues entirely (our examples will remain simple). To keep these *notational labels* separate from *structural atoms*, we will use italics for the former. This notation will handle both shared tails and genuinely circular structures. Thus the structure notated in this extended notation as follows:

```
(+ J: (* 3 4) :J)                                    (S4-620)
```

would be notated graphically as:

(S4-621)



This would be simply, but misleading, printed out in the regular notation as:

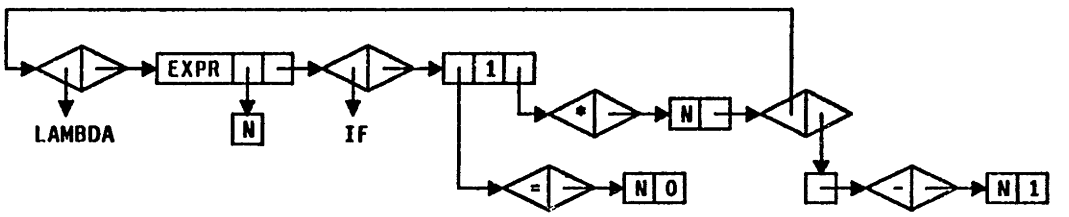
```
(+ (* 3 4) (* 3 4))                                  (S4-622)
```

More interesting is the following *structurally* (token) circular definition of factorial:

```
F: (LAMBDA EXPR [N]                                  (S4-623)
     (IF (= N 0)
         1
         (* N (:F (- N 1)))))
```

which has the following graphical notation:

(S4-624)



If we printed this out in the regular notation, it would lead to the following infinite lexical item:

(S4-625)

```
(LAMBDA EXPR [N]
   (IF (= N 0)
       1
       (* N ((LAMBDA EXPR [N]
                (IF (= N 0)
                    1
                    (* N ((LAMBDA EXPR [N]
                             ... )
                          (- N 1))))
             (- N 1))))))
```

The first intuition, then, is to make the type-equivalent parts of the definition of the various Y operators in fact identical. We start with Y_1. The original definition was this:

(S4-626)

```
(DEFINE Y1
   (LAMBDA EXPR [FUN]
      ((LAMBDA EXPR [X] (FUN (X X)))
       (LAMBDA EXPR [X] (FUN (X X))))))
```

Collapsing the largest type-equivalent structures yields:

(S4-627)

```
(DEFINE Y1
   (LAMBDA EXPR [FUN]
      (K: (LAMBDA EXPR [X] (FUN (X X)))
          :K)))
```

No particular mileage is obtained, however, since this remains as non-terminating as ever. We similarly had this definition of Y_2:

(S4-628)

```
(DEFINE Y2
   (LAMBDA EXPR [FUN]
      ((LAMBDA EXPR [X] (FUN (X X)))
       (LAMBDA EXPR [X]
          (LAMBDA EXPR ARGS
             ((FUN (X X)) . ARGS))))))
```

The two main `(LAMBDA EXPR [X] ... )` terms are different, because of the processing deferral, but observe that there is no *need* for the first of these to be different; thus the following is

essentially similar:

```
(DEFINE Y2a                                                    (S4=629)
   (LAMBDA EXPR [FUN]
      ((LAMBDA EXPR [X]
          (LAMBDA EXPR ARGS
             ((FUN (X X)) . ARGS)))
       (LAMBDA EXPR [X]
          (LAMBDA EXPR ARGS
             ((FUN (X X)) . ARGS))))))
```

We can now collapse this:

```
(DEFINE Y2b                                                    (S4=630)
   (LAMBDA EXPR [FUN]
      (G: (LAMBDA EXPR [X]
             (LAMBDA EXPR ARGS
                ((FUN (X X)) . ARGS)))
       :G))
```

But now it becomes natural to begin to collapse the reductions — to do the implied β-reductions. In other words we are attempting to minimise the structure in Y_2: collapsing type-equivalent structure does part of that task; applying β-reductions in non-applicative order, where possible, helps as well. In particular, X will be bound to the structure labelled G, and the reduction of G with itself happens directly. Thus we get:

```
(DEFINE Y2c                                                    (S4=631)
   (LAMBDA EXPR [FUN]
      (G: (LAMBDA EXPR [X]
             (LAMBDA EXPR ARGS
                ((FUN (:G :G)) . ARGS)))
       :G))
```

But this whole thing can collapse, since it is the application of G to G that we are concerned with:

```
(DEFINE Y2d                                                    (S4=632)
   (LAMBDA EXPR [FUN]
      (K: (LAMBDA EXPR ARGS
             ((FUN :K) . ARGS)))))
```

Furthermore, we can go back to the original style whereby the first application does not wait for arguments. We will call this Y_3 for discussion, since it is counts as a distinct version:

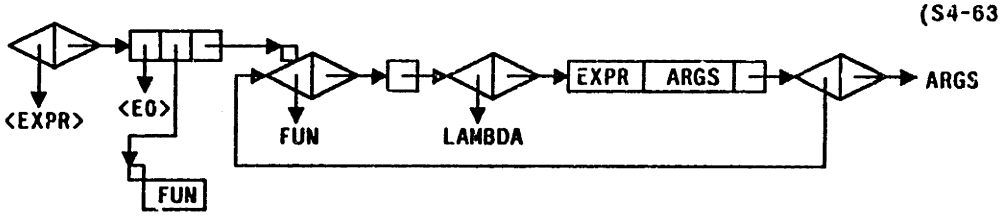
```
(DEFINE Y3                                                     (S4-633)
   (LAMBDA EXPR [FUN]
      K: (FUN (LAMBDA EXPR ARGS
                 (:K . ARGS)))))
```

The normal form version of Y_3 is thus:

```
(<EXPR> E0 '[FUN] 'K:(FUN (LAMBDA EXPR ARGS (:K . ARGS))))          (S4-634)
```

This closure has the following graphical notation:

(S4-636)



Y_3 is again an adequate Y operator, both declaratively and procedurally, for any singly-recursive extensional procedure. ((Y3 H) 6) can be shown to designate 720, and it will return 720. But there is something still a little odd. The inherent circularity in the Y operator has been brought out; by collapsing type-identities onto individual identities, we have shown that the Y operator is *itself*, in the deepest sense, circular. In other words we have seen that the definitions of Y_1 and Y_2 are *themselves* ***type-circular-deferred***; Y_3, in contrast, is ***token circular***. This result is useful, because it shows us how type-identity has covered for a lack of primitive circularity in a tree-structured formalism. Furthermore, it is evident that if a function is in a deep sense circular, it may be able to engender essentially circular behaviour from a non-circular argument. But it would be more to the point if we could show another version of this same thing: how all of these Y operators, Y_1, Y_2, and Y_3 generate *type-circular closures*. Additionally, it would be more powerful if we can define still another version of the Y operator Y_4 that generated *token-circular closures*. We needn't care how Y *itself* is structured: we ought to be more interested in the structure of the procedures Y returns.

We will turn, therefore, to an examination of the intensional form of the procedures that each version of Y returns. We will initially use H as our example, and will look first at our latest version, Y_3. We begin with:

```
(Y3 H)                                                              (S4-636)
```

Substituting the normal-form closure of Y_3 from S4-634:

```
((<EXPR> E0 '[FUN] 'K1: (FUN (LAMBDA EXPR ARGS (:K1 . ARGS))))      (S4-637)
 H)
```

Similarly we can expand the binding of H, since we are about to reduce an EXPR:

```
((<EXPR> E0 '[FUN] ' K1: (FUN (LAMBDA EXPR ARGS (:K1 . ARGS))))      (S4-638)
 (<EXPR> E0 '[FACT] '(LAMBDA EXPR [N]
                       (IF (= N 0) 1 (* N (FACT (- N 1)))))))
```

Binding FUN to the second EXPR closure, and substituting that into the body, we get the following (this is slightly subtle, because we interpret through the pair that we have notated with K; nonetheless the internal use of it retains the full pair, as indicated):

```
((<EXPR> E0 '[FACT] '(LAMBDA EXPR [N]                                (S4-639)
                       (IF (= N 0) 1 (* N (FACT (- N 1))))))
 (LAMBDA EXPR ARGS (K1: (FUN (LAMBDA EXPR ARGS) (:K1 . ARGS))
                    . ARGS)))
```

Once again, noting that the reduction is of an EXPR, we can normalise the single argument. This normalisation happens in an environment which is like E_0 except extended so that FUN is bound to the closure to which H expanded::

```
(K2:(<EXPR> E0 '[FACT] '(LAMBDA EXPR [N]                             (S4-640)
                          (IF (= N 0) 1 (* N (FACT (- N 1))))))
 (<EXPR> [['FUN :K2] ... E0]
         'ARGS
         '(K1: (FUN (LAMBDA EXPR ARGS) (:K1 . ARGS)) . ARGS)))
```

Finally, we can bind this to FACT, and normalise the body of the closure being reduced, yielding our answer. In this case the crucial thing is the binding of FACT, so we illustrate the expanded environment:

```
(<EXPR> [['FACT (<EXPR> [['FUN '(<EXPR> E0                           (S4-641)
                                        '[FACT]
                                        '(LAMBDA EXPR [N] :K3))]
                           ... E0]
                         'ARGS
                         '(K1: (FUN (LAMBDA EXPR ARGS) (:K1 . ARGS))
                           . ARGS)]
           ... E0]
         '[N]
         'K3: (IF (= N 0) 1 (* N (FACT (- N 1)))))
```

Though we need not go through them in detail, expansions for the other three varieties of Y can be worked out in the same fashion. We end up with the results summarised in the following illustrations. Rather than use a particular H we have generalised these results to use a generic single-argument function of form:

```
<H>  ≡  (LAMBDA EXPR [<LABEL>] <FORM>)                               (S4-642)
```

where for the time being we assume

```
<FORM>  ≡  (LAMBDA EXPR <PATTERN> <BODY>))                    (S4-643)
```

(This is limiting, and we will generalise it presently.) First we look at Y_1. Although $(Y_1\ H)$ would never return, it is easy enough to work out what it would return if we waited until the end of time. A few steps through the reduction shows immediately the structure of the infinite descent. We start with

```
(Y1 H)                                                        (S4-644)
```

which is equivalent to

```
((LAMBDA EXPR [FUN]                                           (S4-645)
    ((LAMBDA EXPR [X] (FUN (X X)))
     (LAMDBA EXPR [X] (FUN (X X)))))
 H)
```

Binding FUN to H and reducing yields (underlining is used to indicate inverse normalisation):

```
((LAMBDA EXPR [X] (H (X X)))                                  (S4-646)
 (LAMBDA EXPR [X] (H (X X))))
```

Another reduction:

```
(H ((LAMBDA EXPR [X] (H (X X)))                               (S4-647)
    (LAMBDA EXPR [X] (H (X X)))))
```

But the argument to H here is type-equivalent to S4-646; hence it is apparent that $(Y_1\ H)$ will generate more and more of:

```
(H (H (H ( ... (Y1 H)...))))                                  (S4-648)
```

If we were to collapse type-equivalences into token identities, and thereby terminate this infinite process, we would have:

```
K: (H :K)                                                     (S4-649)
```

But this cannot be posited as the appropriate infinite closure, since it is not in normal form. Expanding the H yields

```
(Y1 H)   ⇒   K: ((<EXPR> <ENV> '[<LABEL>] '<FORM>) :K)        (S4-650)
```

If we do one more reduction by hand, we bind <LABEL> to K and normalised <FORM>. Assuming the structures of S4-643 apparently yields:

```
(Y1 H)   ⇒   (<EXPR> [['<LABEL>                                          (S4-651)
                       'K:((<EXPR> <ENV> '[<LABEL>] '<FORM>) :K)]
                       ... <ENV>]
                      '<PATTERN>
                      '<BODY>)
```

But the point of K was to label the result, *whatever it was that we decided* $(Y_1\ H)$ *returned*; thus this should really be:

```
(Y1 H)   ⇒   K: (<EXPR> [['<LABEL> :K] ... <ENV>]                       (S4-652)
                        '<PATTERN>
                        '<BODY>)
```

This is in a sense ideal; the problem is that it is the result of an infinite computation. We will take it up below, however, after looking at Y_2 and Y_3.

A simple view of the result of normalising $(Y_2\ H)$ leads to the following notation, more easily first understood without using our extended notation to indicate shared structure. Note that this is a finite structure because only one generation of the production of the infinite type-equivalent tree has been executed (this is the essence of Y_2 generating *deferred* circular closures).

```
(Y2 H)   ⇒   (<EXPR>                                                     (S4-653)
              [['<LABEL> '(<EXPR>
                          [['X '(<EXPR>
                                 [['FUN '(<EXPR> E0 '[<LABEL>] '<FORM>)]
                                  ... E0]
                                 '[X]
                                 '(LAMBDA EXPR ARGS ((FUN (X X)) . ARGS)))]
                           ['FUN '(<EXPR> E0 '[<LABEL>] '<FORM>)]
                           ... E0]
                          'ARGS
                          '((FUN (X X)) . ARGS))]
               ... E0]
              '<PATTERN>
              '<BODY>)
```

If we explicitly identify all shared structure we have:

```
(Y2 H)   ⇒   (<EXPR>                                                     (S4-654)
              [['<LABEL> '(<EXPR>
                          [['X '(<EXPR>
                                 [K1:['FUN '(<EXPR> E0 '[<LABEL>] '<FORM>)]
                                  ... E0]
                                 '[X]
                                 '(LAMBDA EXPR ARGS
                                     K2:((FUN (X X)) . ARGS)))]
                           :K1 ... E0]
                          'ARGS
                          ':K2)]
               ... E0]
```

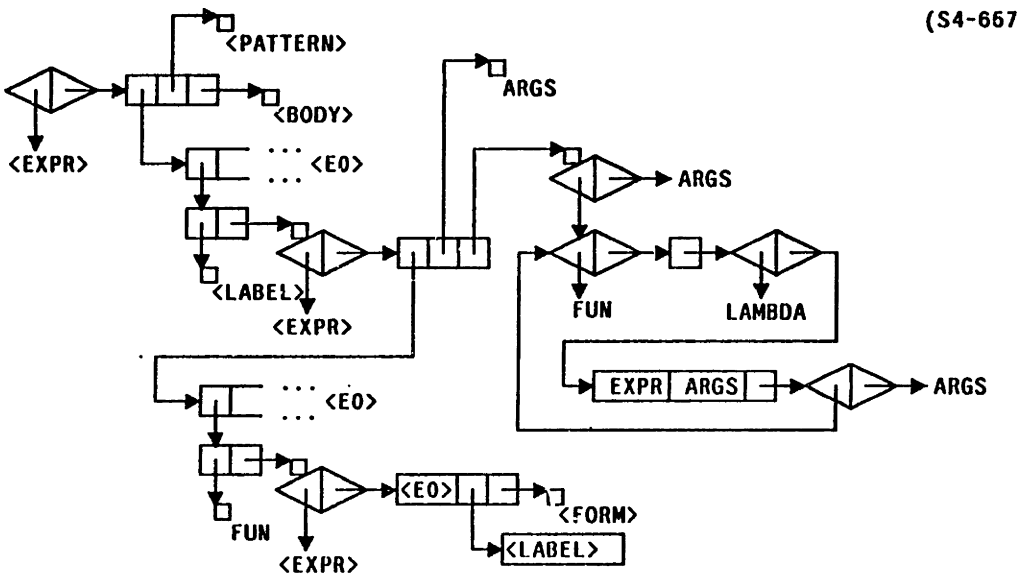
```
                '<PATTERN>
                '<BODY>)
```

Finally, the result obtained above in S4-641, re-written in terms of abstract <PATTERN> and <BODY>:

```
(Y3 H)  ⇒  (<EXPR> [['<LABEL>                                          (S4-666)
                    '(<EXPR> [['FUN '(<EXPR> E0 '[<LABEL>] '<FORM>)]
                              ... E0]
                             'ARGS
                             '(K1: (FUN (LAMBDA EXPR ARGS (:K1 . ARGS)))
                                . ARGS)]
                   ... E0]
                  '<PATTERN>
                  '<BODY>)
```

In graphical form:

(S4-667)

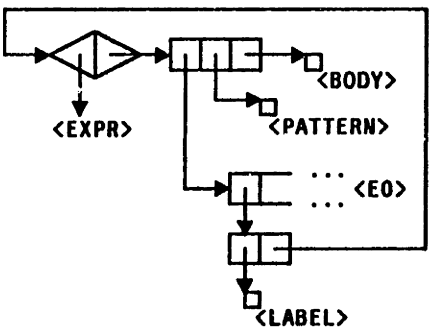


From this last it becomes plain how both $(Y_2\ H)$ and $(Y_3\ H)$ fail to be what we want. If we were to achieve token circularity in the resultant closure, the binding of FACT in the closed environment *would be the overall closure*, not — like in S4-667 — a closure that would engender a type-equivalent closure. The behaviour we aim for, in other words, is that produced after an infinite amount of processing by Y_1. The question is whether it is possible to define a version of Y called Y_4 that would be procedurally finite but computationally equivalent in result:

```
(Y4 <H>) ⇒  K: (<EXPR> [['<LABEL> :K] ... E0]                          (S4-668)
                       '<PATTERN>
                       '<BODY>)
```

In graphical form would be this:

(S4-659)



This is particularly mandated because Y_1 (and hence by assumption Y_4) works correctly with intensional procedures, whereas neither Y_2 nor Y_3 have that advantage. The problem is that both Y_2 and Y_3 yield processing-deferred closures, of the form (actually the normalisation of this)

```
(LAMBDA EXPR ARGS (... (FUN . ARGS)))                    (S4-660)
```

which clearly normalises the arguments automatically. It would be possible to complicate the definition so that $(Y_2\ H)$ would return

```
(LAMBDA IMPR ARGS (... (FUN . ARGS)))                    (S4-661)
```

just in case FUN was bound to an IMPR, but there is no help in this, since FUN will then be given 'ARGS as its single bound argument, rather the the handles on the arguments intended. (This is a re-occurence of the problem we encountered with IF.) No obvious solution presents itself.

There are many reasons, then, pushing us towards a tractable definition of Y_4. If we could assume that the <FORM> term in all H expressions was, as suggested in S4-643, the following:

```
<FORM>  =  (LAMBDA EXPR <PATTERN> <BODY>)                (S4-662)
```

then an adequate, if ugly, Y_4 would be easy enough to define; a single-function version is the following:

```
(DEFINE Y4a                                              (S4-663)
   (LAMBDA EXPR [FUN]
      (LET [[CLOSURE ↑(FUN '?)]]
         (BLOCK (RPLACN 2 (1ST ↑(1ST (CDR CLOSURE))) ↑CLOSURE)
                ↓CLOSURE))))
```

This works by brute force: the variable CLOSURE is bound to a designator of the closure *qua* pair; (1ST (CDR CLOSURE)) selects the internal environment designator; ↑(1ST (CDR CLOSURE)) obtains access to this *as a rail* (environments are sequences; therefore we need an uparrow to designate the rail that in turn designates the environment). The first element of this rail is the binding of the procedure's name; it will temporarily be bound to the atom "?", in virtue of the term (FUN '?) in the third line. The term ↑CLOSURE, finally, designates a handle that in turn designates the closure; since we are dealing with the environment designator as an explicit rail, we need to insert the appropriate closure designator as a binding. We are working, in other words, *two* levels above the object level at which the procedure will ultimately be used. Note that there is an approximate balance in the use of an up-arrow in both of the arguments to RPLACN. Finally, the closure itself cannot be designated as a result; Y_{4a} applied to a function should yield a function; hence ↓CLOSURE is the exit form.

However one of our whole reasons for constructing an explicit and adequate Y operator is to handle a wider variety of forms. Suppose that we executed this:

```
(Y4a (LAMBDA EXPR [FIBONNACCI]                                    (S4-664)
       (LET [[FIB-1 1]
             [FIB-2 1]]
         (LAMBDA EXPR [N]
           (COND [(= N 1) FIB-1]
                 [(= N 2) FIB-2]
                 [$T (+ (FIBONNACCI (- N 1))
                        (FIBONNACCI (- N 2)))]))))))
```

Without spelling out the details of all the intermediate states, it is clear that Y_{4a} as defined above in S4-663 would smash the binding of the procedural constant FIB-2, rather than the binding of FIBONNACI. This is because the form of the closure that would be returned would be this:

```
(<EXPR> [['FIB-2 '1]                                              (S4-665)
         ['FIB-1 '1]
         ['FIBONNACCI '?]
          ... E0]
        '[N]
        '(COND ... ))
```

Thus Y_{4a} cannot be adopted. Given this realisation, a second proposal natural arises: rather that modifying the *first* binding in the environment in the closure, we should modify the binding *of the name* FIBONNACCI. In the next section we will introduce a procedure called

REBIND of three arguments: a variable, a binding, and an environment; it destructively modifies the binding of the variable in the environment so as to be the new binding. In other words we might imagine the following version of Y_4:

```
(DEFINE Y4b                                                    (S4-666)
   (LAMBDA EXPR [FUN]
      (LET [[CLOSURE ↑(FUN '?)]]
         (BLOCK (REBIND '<LABEL> ↑(1ST (CDR CLOSURE)) ↑CLOSURE)
                ↓CLOSURE))))
```

There are however two problems with this. First, Y_4 is not currently cognisant of the name that the H form uses for its internal recursive use (hence the <LABEL> in the preceding code), and it seems inelegant to have to extract it or pass it out explicitly to Y. This could be arranged, however, except that there is a worse problem: there is no guarantee that <LABEL> will be defined. One of the simplest such examples was our illustration in S4-602 of

```
(DEFINE EQ =)                                                  (S4-667)
```

which we said would expand (if we were to adopt this variant) into

```
(SET EQ (Y4b (LAMBDA EXPR [EQ] =)))                            (S4-668)
```

But (LAMBDA EXPR [EQ] =) will return the primitive closure

```
(<EXPR> E0 [A B] (= A B))                                      (S4-669)
```

and the subsequent call to REBIND will fail to find EQ bound in E_0. Nor should it *add* such a binding. But worse, it should not *necessarily* rebind the first occurence of <LABEL>; if the H procedure did not bind it, then there might be a *different* use of <LABEL> in some encompassing context, and it would be disastrous to modify that. Hence Y_{4b} must be rejected as well.

Fortunately, all is not lost: two further variants enable us to side-step most of these difficulties. They hinge on the same observation: the REBIND in Y_{4b} and the RPLACN in Y_{4a} were similar in intent: they both attempted to locate and modify the appropriate binding in the environment of the returned closure. The problem with both was a potential error in determining the appropriate binding. The suggestion then is to ask whether we could obtain any better access to that binding, rather than attempting to discover, after the fact, which binding it was. There are two answers to this, one of which we can implement, one requiring a facility not currently part of 2-LISP.

The first, to be embodied in Y_{4c}, is this: we (that is to say the code constituting Y_4) pass to the H function a temporary binding, currently the dummy atom "?". Suppose instead we were to generate a temporary closure, and then to modify *that very closure* when the H function returned. Since we cannot actually change what closure it is (what *pair* it is, since pairs are used for closures), we would have to use RPLACA and RPLACD. The definition of Y_{4c} is as follows:

```
(DEFINE Y4c                                                    (S4-670)
   (LAMBDA EXPR [FUN]
      (LET* [[TEMP (LAMBDA EXPR ARGS (ERROR))]
             [CLOSURE ↑(FUN TEMP)]]
         (BLOCK (RPLACA ↑TEMP (CAR CLOSURE))
                (RPLACD ↑TEMP (CDR CLOSURE))
                TEMP))))
```

Though this cannot be said to be elegant, it is no less elegant than Y_{4a} or Y_{4b}. Note, furthermore, how it solves the problems that plague the previous two versions. If CLOSURE does not bind TEMP, the side effects engendered by the first two BLOCK expressions will be discarded upon exit from Y_{4c}, since all access to TEMP will be thrown away. If TEMP *is* bound, somewhere in the environment within CLOSURE, then that binding will be to a closure that is *changed* to be equivalent to the closure returned by applying H to "itself".

The crucial word here is "equivalent". What makes Y_{4c} work is that it, too, trades on a type-equivalence. It makes TEMP be *type-equivalent* to CLOSURE, rather than actually identical. Actual token identity eludes us, since we have no way of obtaining the information of where the occurence of TEMP is, and no primitive structural procedure enabling us to change that pair to actually be a different one.

This does, however, lead to a fourth suggestion. Suppose there were a primitive 2-LISP procedure called REPLACE that generalised the abilities provided by RPLACA, RPLACD, RPLACT, and RPLACN. The idea would be that (RPLACE <X> <Y>) would affect the field so that all occurences of <X> were hencefforth occurences of <Y>. REPLACE should be restricted to the "pseudo-composite" structure types: pairs and rails (no sense can really be made of actually replacing constants). This behaviour is very similar to our provision of a RPLACT that works with a zero index (indeed, the implementational consequences are very similar: so-called "invisible pointers" would be required on pairs as well as rails, in a natural implementation on a Von-Neuman-like machine). The four present structural side-effect procedures could be defined in terms of REPLACE as follows:

```
(DEFINE RPLACA                                                    (S4-671)
   (LAMBDA EXPR [PAIR NEW-CAR]
      (REPLACE PAIR (PCONS NEW-CAR (CDR PAIR)))))

(DEFINE RPLACD                                                    (S4-672)
   (LAMBDA EXPR [PAIR NEW-CDR]
      (REPLACE PAIR (PCONS (CAR PAIR) NEW-CDR))))

(DEFINE RPLACT                                                    (S4-673)
   (LAMBDA EXPR [INDEX RAIL NEW-TAIL]
      (REPLACE (TAIL INDEX RAIL) NEW-TAIL)))

(DEFINE RPLACN                                                    (S4-674)
   (LAMBDA EXPR [INDEX RAIL NEW-ELEMENT]
      (REPLACE (TAIL (- INDEX 1) RAIL)
               (PREP NEW-ELEMENT (TAIL INDEX RAIL)))))
```

Such a REPLACE would facilitate the following definition of Y_{4d}:

```
(DEFINE Y4d                                                       (S4-675)
   (LAMBDA EXPR [FUN]
      (LET* [[TEMP (LAMBDA EXPR ARGS (ERROR))]
             [CLOSURE ↑(FUN TEMP)]]
        (BLOCK (REPLACE ↑TEMP CLOSURE)
               TEMP))))
```

Since we do not have such a REPLACE, we will have to adopt the marginally less satisfactory Y_{4c}.

Since the circular closures that Y_4 constructs seem not unlike what a simple approach might have suggested, the reader may question our long diversion through three other versions of the Y operator. However our investigation can be defended on a variety of counts. First, Y_{4c} is by no means isomorphic to the standard approach, as the discussion at the begining of this section argued, and as the discussion of such procedures as PROTECTING in the next section will emphasize. Furthermore, Y_{4c} is similar in external behaviour to the Y_2 we based relatively directly on Church's fixed point operator; thus we can use Y_4 in all of the situations we used Y_2. Embedded definitions, "own-variables" like those illustrated in S4-610 and S4-612 are still supported, and so forth — all capabilities beyond those provided in standard LISPs. If we had started with a primitive "DEFINE" operator we would likely not have provided these capabilities, and even if we had we would have had to defend them *ex post facto*, rather than seeing how they arise naturally out of the very nature of recursive definitions. Second, we still have a facility for treating recursive definitions that is not primitive, even though our final version is superficially inelegant. It is worth noting, however, whence this inelegance arises. The closures that Y_{4c} constructs are

much *more* elegant than those generated by the simpler Y_2; the only reason that Y_{4c} is messy is that it is messy to construct circular structure in a fundamentally serial dialect. The awkwardness of S4-670, in other words, emerges from the fact that it is awkward in an essentially tree-oriented formalism to construct non-tree-like structures. There is no *inherent* lack of cleanliness either in the closures constructed, or in the abstract task of constructing them. In a radically different calculus, based on a more complete topological mapping between program and behaviour, such construction would be essentially trivial. (It must be admitted, however, that because token circular closures are not primitively provided, our definition of Y_4 required meta-structural access, whereas the simpler Y_2 did not: it was purely extensional.)

Third, we have seen how the question of providing public access to the names of recursive procedures, and the question of providing a self-referential name to be used within a recursive definition, are at heart different issues. The macro definition of DEFINE given in S4-599, can be carried over essentially intact so as to use Y_{4c}. In particular, expressions of the form

```
(DEFINE <LABEL>                                                    (S4-676)
   (LAMBDA <TYPE> <PATTERN> <BODY>))
```

will be taken to be abbreviatory for

```
(SET <LABEL>                                                       (S4-677)
     (Y4c (LAMBDA EXPR [<LABEL>]
            (LAMBDA <TYPE> <PATTERN> <BODY>)))
```

Nothing crucial any longer depends on the LAMBDA form of the second argument to DEFINE, however, so we can generalise this; expressions of the form

```
(DEFINE <LABEL> <FORM>)                                            (S4-678)
```

will be taken to abbreviate:

```
(SET <LABEL> (Y4c (LAMBDA EXPR [<LABEL>] <FORM>)))                 (S4-679)
```

Again, we will examine the import of the (SET ...) in the next section.

Fourth, our developmental approach has shown us how the original Y operator, and our side-effect engendering closure modifier, are essentially related. Both have to do with a kind of self-use, one effected in virtue of a type-equivalence, one effected in virtue of a circular path for the processor.

It is appropriate to review our four flavours of Y in light of our new terminology and conclusions:

- Y_1: This version is itself type-circular deferred; it generates type-circular (but not deferred) closures. Recommending Y_1 is the fact that it is the direct embodiment of the standard fixed-point function in the meta-language; ruling it out, however, is the fact that it requires an infinite amount of time to generate its closures (because they are not deferred).
- Y_2: Type-circular deferred itself, Y_2 also generates type-circular deferred closures; as such it is procedurally tractable, but inefficient of both time and structure.
- Y_3: Y_3 is itself token-circular, but, like Y_3, it generates type-circular-deferred closures. Thus, although of some interest, it had little to recommend it beyond Y_2.
- Y_4: Y_4 (in all of its versions) was not itself circular at all, but it generates token-circular closures. Although it was not singularly elegant in this construction, the closures that resulted were considered optimal, and thus it was selected.

There are some tidying-up details we need to attend to before moving on to a study of environments and variables. First, there is a question of terminology. Though Y_{4c} is developmentally related to the original Y operator, as our discussion has shown, and though it designates the same function, it is markedly different intensionally; it is not strictly fair, in other words, to call it by Church's name "Y". Since it is the only version we will adopt in 2-LISP, it would be unnatural to retain the "4c" subscript. In the following definitions, therefore, and throughout the remainder of the dissertation, we will use the name "Z" for this procedure.

The single argument Z of S4-670 can of course be generalised to handle multiple mutually recursive definitions, in very much the way we generalised Y_2. The following is a two-procedure version (we won't use this, but it leads towards the subsequent definition of Z*):

```
(DEFINE ZZ                                                    (S4-680)
   (LAMBDA EXPR [FUN1 FUN2]
      (LET* [[TEMP1 (LAMBDA EXPR ARGS (ERROR))]
             [TEMP2 (LAMBDA EXPR ARGS (ERROR))]]
            [CLOSURE1 ↑(FUN1 TEMP1 TEMP2)]
            [CLOSURE2 ↑(FUN2 TEMP1 TEMP2)]]
         (BLOCK (RPLACA ↑TEMP1 (CAR CLOSURE1))
                (RPLACD ↑TEMP1 (CDR CLOSURE2))
                (RPLACA ↑TEMP2 (CAR CLOSURE1))
                (RPLACD ↑TEMP2 (CDR CLOSURE2))
                [TEMP1 TEMP2]))))
```

Finally, a version able to handle an indefinite number of mutually recursive definitions:

```
(DEFINE Z*                                                          (S4-681)
   (LAMBDA EXPR FUNS
      (LET* [[TEMPS (MAP (LAMBDA EXPR ?
                           (LAMBDA EXPR ARGS (ERROR)))
                         FUNS)]
             [CLOSURES ↑(MAP (LAMBDA EXPR [FUN] (FUN . TEMPS)) FUNS)]]
         (MAP (LAMBDA EXPR [TEMP CLOSURE]
                (BLOCK (RPLACA ↑TEMP (CAR CLOSURE))
                       (RPLACD ↑TEMP (CDR CLOSURE))
                       TEMP))
              TEMPS
              CLOSURES))))
```

The behaviour is essentially similar to that of Z and ZZ.

In 3-LISP both Z and Z* will be structurally modified in very minor ways, but in behaviour and semantics they will remain essentially unchanged. Finally, that Z satisfies our original goals — supporting general recursion, mutual recursion, EXPRs and IMPRs, embedded definitions, own variables, and so forth — is shown in the following set of examples, by way of review. First, a paradigmatic recursive definition:

```
(Z (LAMBDA EXPR [FACT]                                              (S4-682)
     (LAMBDA EXPR [N]
        (IF (= N 0)
            1
            (* N (FACT (- N 1)))))))

    ⇒   K: (<EXPR> [['FACT K:] ... E0] '[N] (IF (= N 0) 1 ... ))
```

Second, Z used on a non-recursive definition does not introduce trouble:

```
(Z (LAMBDA EXPR [EQ] =)                                             (S4-683)

    ⇒   (<EXPR> E0 '[A B] (= A B))
```

Third, Z supports embedded definitions:

```
(LET [[X 1]]                                                        (S4-684)
   (Z (LAMBDA EXPR [FACT]
        (LAMBDA EXPR [W] (+ X W)))))

    ⇒   K: (<EXPR> [[FACT :K]['X '1] ... E0] '[W] '(+ X W))
```

Fourth, it supports "own" variables:

```
(Z (LAMBDA EXPR [FIBONNACCI]                                        (S4-685)
     (LET [[FIB-1 1]
           [FIB-2 1]]
```

```
(LAMBDA EXPR [N]
   (COND [(= N 1) FIB-1]
         [(= N 2) FIB-2]
         [$T (+ (FIBONNACCI (- N 1))
                (FIBONNACCI (- N 2)))]))))))

⇒  K: (<EXPR> [['FIB-2 '1]
               ['FIB-1 '1]
               ['FIBONNACCI :K]
               ... E0]
              '[N]
              '(COND [(= N 1) FIB-1] ... ))
```

Fifth, Z supports intensional procedures, recursive as well as non-recursive:

```
(Z (LAMBDA EXPR [IF]                                              (S4-686)
     (LAMBDA IMPR [PREMISE T-CONSEQUENT F-CONSEQUENT]
       (IF (= '$T (NORMALISE PREMISE))
           (NORMALISE T-CONSEQUENT)
           (NORMALISE F-CONSEQUENT)))))

⇒  K: (<IMPR> [['IF :K] ... E0]
              '[PREMISE T-CONSEQUENT F-CONSEQUENT]
              '(IF ... ))
```

Sixth and finally, Z* may be used for non-top-level mutually recursive procedures (this is an expansion of what normally be written using the abbreviatory LABELS):

```
(LET [[X 3] [Y 4]]                                                (S4-687)
  (LET [[[G1 G2]
         (Z* (LAMBDA EXPR [G1 G2]
               (LAMBDA EXPR [X Y] ((G2 X Y) (+ X Y) X)))
             (LAMBDA EXPR [G1 G2]
               (LAMBDA EXPR [X Y] (IF (EVEN (* X Y)) - G1)))))]]
    [G1 G2]))

⇒  [K1: (<EXPR> [['G1 ':K1]['G2 ':K2] K3::['X '3]['Y '4] ... E0]
                '[X Y]
                '((G2 X Y) (+ X Y) X))
    K2: (<EXPR> [['G1 ':K1]['G2 ':K2] ::K3]
                '[X Y]
                '((G2 X Y) (+ X Y) X))]
```

By "$K_3::$" in this example we mean to label the *tail* of a rail; by "$::K_3$" we mean that the tail is the rail so notated. The two closures, in other words, share their second tail (why this is so will be explained in the next section). Given this sharing, it might seem that it would be more economic if Z* could construct a single environment, engendering something like the following:

```
(LET [[X 3] [Y 4]]                                                    (S4-688)
   (LET [[[G1 G2]
          (Z* (LAMBDA EXPR [G1 G2]
                (LAMBDA EXPR [X Y] ((G2 X Y) (+ X Y) X)))
              (LAMBDA EXPR [G1 G2]
                (LAMBDA EXPR [X Y] (IF (EVEN (* X Y)) - G1))))]]
     [G1 G2]))

   ⇒   [K1: (<EXPR> K3:[['G1 ':K1]['G2 ':K2]['X '3]['Y '4] ... E0]
                    '[X Y]
                    '((G2 X Y) (+ X Y) X))
        K2: (<EXPR> :K3                                  ; This is wrong!!
                    '[X Y]
                    '((G2 X Y) (+ X Y) X))]
```

However this is incorrect: it is an artefact of the simplicity of the two H functions given to Z* that their environments are in this case type-equivalent. All of the arguments against Y_{4a} and Y_{4b} would rule out any such simplification.

4.c.vi. Environments and the Setting of Variables

One issue remains: the (destructive) setting of variables. In the previous part of section 4.c we have discharged recursion, and reduced the question of defining procedures to the question of setting of variables. We know that environments are the recorders of variable bindings, and we know how to use LAMBDA terms to bind names in static contexts, but there are two related questions we have not yet considered. First, although we have used the primitive SET a variety of times, we have not explained it. Second, although we have shown how rails of handles are normal-form environment designators, we have not questioned the *identity* of those environment-encoding rails. As usual, these are two sides of the same coin: it is the impact of structural side-effects that determines, and depends on, the identity of the structures in question.

We can define whatever behaviour we like; the issue is to determine what makes sense. This is particularly difficult in a statically scoped system; as Steele and Sussman have made clear, there is an inherent tension between statically scoped dialects and the ability to dynamically affect the procedures bound to previously used atoms. The standard exemplar of this tension is in the "top-level" READ-NORMALISE-PRINT loop (our version of READ-EVAL-PRINT, of course) with which a user interacts with the language processor, since it is in this context that procedure definitions are typically altered. It is not impossible to construct a user interface that binds variables using standard LAMBDA binding protocols; the form (SET <X> <Y>), for example, when used at top level, could be treated as a macro form that expanded to (LET [[<X> <Y>]] (READ-NORMALISE-PRINT)). The problem is that this is not the behaviour one wants: typically when re-defining procedures, as we will see below, one wants the re-definition to affect other previously defined procedures, which this suggestion would not engender. The question is not one of how to support user interfaces, but one of how to provide controlled protocols for effecting change on extant structure.

(In point of fact, that last characterisation is too broad: we already have the four versions of RPLAC- for changing *structural field elements.* Our present concern is with changing *environments.* Once put this way, it is natural to ask about changing *continuations,* since those three entities constitute an entire context. Modifying continuations, however, is a matter we will defer until chapter 5.)

It is important to realise that there is nothing *unique* about user interaction in this regard; the user interface is merely the place where side-effects to environments are most typically requested. Any program able to modify its own structures — any substantially reflective process, in other words — will encounter the same issues. It was one of the mandates on reflection we set out in chapter 1 that a reflective process must have *causal access* to the structures in terms of which it behaves, in order that the reflective abilities may matter. One of the ways mattering manifests itself is in the ability to modify bindings in various contexts. In this present chapter, since we have not yet taken up reflection as its own subject, we will constrain our examples primarily to user-interaction, but the reader should be aware that this is by way of example only.

As usual, the best answer to this set of problems — the simple provision of a clear facility that rationalises static scoping and dynamic control — will emerge from the reflective abilities in 3-LISP. We do not yet have access to this machinery, but we will present an inchoate version of it in 2-LISP; one that makes use of some emergent reflective properties we have already accepted as part of our definition of closures. In particular, we will use some code that changes *environment designators*, rather than simply providing primitives that affect the processor's environment. A hint of this approach was given in section 4.c.ii, where we warned that the use of a procedure called REBIND on the environment designator contained within a closure could affect the semantics of that closure. This equivocation between accessing environments directly, or indirectly through environment designators, is part of 2-LISP's inelegance. But so be it.

In describing closures we showed the form of environment designators; if we add the assumption that these environment designators not only *encode* the environment used by the processor, but actually *causally embody it* in such a way that changing them will affect subsequent processor behaviour (very definitely an additional assumption, but one that we tacitly adopted when we set out the definition of z), then we can simply define a simple variable setting procedure. We call our procedure REBIND; it takes three arguments — a variable (atom), a binding, and an environment. A tentative definition is the following (we discuss what should happen when the variable is not bound in a moment):

```
(DEFINE REBIND                                                    (S4-691)
   (LAMBDA EXPR [VARIABLE BINDING ENVIRONMENT]
      (COND [(EMPTY ENVIRONMENT) <the variable isn't bound>]
            [(= VARIABLE (1ST (1ST ENVIRONMENT)))
             (RPLACN 2 ↑(1ST ENVIRONMENT) ↑BINDING)]
            [$T (REBIND VARIABLE BINDING (TAIL 1 ENVIRONMENT))])))
```

The up-arrows are necessary because environments are sequences of sequences, and RPLACN requires that its arguments designate rails.

The most important property of this definition is the implicit semantic flatness of its arguments. The function designated by this definition is to be applied to variables, bindings, and environments, in a perfectly straightforward fashion. REBIND is extensional: it should therefore be called with arguments expressions that *designate* these three kinds of entity. Variables are atoms; therefore the first argument expression should designate an atom. Bindings are s-expressions, and therefore the second argument expression should designate an s-expression. Similarly, environments are sequences; the third argument should designate a sequence.

If this seems obvious, it is striking to compare it with the behaviour of SET and SETQ in traditional dialects. In particular, it is suddenly becomes clear why in 1-LISP and related dialects the primitive SETQ is natural and common, whereas SET is rare and often awkward: *SETQ is semantically balanced, in that both arguments are at the same semantic level, whereas the 1-LISP SET is unbalanced: the expressions are at different semantic levels.* In order to see this, suppose we wish to set the variable X to be bound to 3 in some environment E. We intend, in other words, to be able, in the context that results, to use X to designate the third natural number (after the binding has happened, (+ X X) should designate six). In 1-LISP we would have the following:

```
> (SETQ X 3)          ; This is 1-LISP                            (S4-692)
> 3
> (+ X X)
> 6
```

Our definition of REBIND, above, is of course rather different. The following is improper:

```
> (REBIND 'X 3 E)                                                 (S4-693)
<ERROR: REBIND, expecting an s-expression, found the number 3)
```

Instead we need to use this:

```
> (REBIND 'X '3 E)                                        (S4-694)
> 3
> (+ X X)
> 6
```

The reason is that the REBIND redex must *mention* both variable and binding: the variable is the atom; the binding is the numeral. Thus REBIND is semantically flat, as it should be. 1-LISP's SETQ is *also* semantically flat (to the extent we can say that anything having to do with evaluation is flat), in that *both expressions are written down in a way that looks as if they are being used.* SETQ, in other words, is more like LAMBDA or LET; the variable argument — the first argument — is like a *schematic designator*, rather than a designator *of* a variable. The result of the binding is such that using the variable will be (designationally) equivalent to using the second argument.

It is for this reason that we have called the 2-LISP version of SETQ by the name "SET", omitting the "Q", so as to rid ourselves of the semantic level-crossing anomaly suggested in the 1-LISP version of the simpler label. SET can approximately be defined in terms of REBIND (we will explain the "approximately" below):

```
(DEFINE SET                                               (S4-695)
   (LAMBDA IMPR [VAR BINDING]
      (REBIND VAR (NORMALISE BINDING) <E>)))
```

Numerous questions have to be answered here: what term should be used for <E> to designate the appropriate environment, and how NORMALISE behaves. The first can properly be answered only in a reflective system; the second will be explained in the next section. However we can depend on one salient fact about NORMALISE: calls to NORMALISE, like calls to every procedure in the entire formalism, are semantically flat: hence (NORMALISE '3) will return '3; (NORMALISE '(+ 2 3)) will return '5. As a consequence, the behaviour engendered by S4-695 is just correct: the variable VAR will be bound to a *designator* of the variable in question, and the parameter BINDING will be bound to a *designator* of the unnormalised binding expression. The explicit call to NORMALISE will return a designator of the expression to which that second argument normalises. These are just the two arguments that we need to give to REBIND.

This definition should make clear a very important fact: what distinguishes the first and second position in a use of SET is just what distinguishes the parameters and body from the arguments in a use of LAMBDA or LET: one set is used *schematically* or *potentially*; the

other *extensionally* or *actually*. They differ in whether they are *processed*, but they do not differ in *semantic level*. Our tearing apart of *evaluation* into *normalising* and *de-referencing*, in other words, means that there are two ways in which an intensional procedure can treat its arguments; as *un-normalised* or as *mentioned*. SET (and LAMBDA) want to do the first; IMPRS want to do the second. The first honours our goal of maintaining semantical flatness, whereas the second does not.

It would be possible, clearly, to make every procedure in 2-LISP extensional — to dispense with IMPRS altogether, in other words — and to quote (using handles) all arguments to all intensional procedures. With meta-structural powers, in other words, one can subsume intensional argument positions.[7] However this last insight into the use of variables in schematic positions suggests that the cleanliness such a protocol would engender is sometimes at odds with another semantic aesthetic: that certain level correspondences between arguments be maintained. This author does not have a strong view on whether it is better to honour one aesthetic or the other, although the mandate to "maintain semantic flatness" seems closer to natural language (we more typically say "*The person called John*", rather than *"The person called 'John'"*). What does seem clear, however, is that the two variable-binding procedures — SET and LAMBDA (LET) — should be *parallel*. If LAMBDA does not require its pattern to be quoted, then SET should not require *its* "variable" argument to be quoted either. If, on the other hand, we insist on `(SET 'X 3)`, we should equally require `(LAMBDA EXPR '[X] '(+ X 1))`.

We can then illustrate the behaviour of our new SET: it is manifestly similar in spirit to the familiar SETQ of traditional systems (at the moment we simply assume that the environment question is resolved — we will take care of it presently):

```
> (SET X (+ 3 4))                                          (34-696)
> 7
> (SET Y (+ X 10))
> 17
> (/ Y X)
> 2
> (SET ANCIENT-AVIATORS '[ICARUS DAEDALUS])
> '[ICARUS DAEDALUS]
> (NTH 1 ANCIENT-AVIATORS)
> 'ICARUS
> (TYPE (REST ANCIENT-AVIATORS))
> 'RAIL
> (LET [[X 3]]          ; LET and SET are semantically parallel,
     (SET X (+ X 1))    ; which, now that we think about it, surely
     X)                 ; makes sense.
```

```
> 4
```

We have mentioned in earlier discussion that all 2-LISP bindings will be in normal-form. The definition of SET shows how this property is naturally preserved from the very meaning of the procedure. Since any expression of the form (SET X Y) is equivalent to (REBIND 'X ↑Y <ENV>), it is necesarily true that all bindings engendered by the use of this procedure will have the property of being in normal-form. We have already seen how LAMBDA bindings in extensional procedures also maintain this property. It is clear from the definition of REBIND presented in S4-691, however, that nothing in that definition prevents the use of a form — such as for example (REBIND 'X '(+ 2 3) <ENV>) — that will establish a binding of a variable to a non-normal form expression. We are led therefore to expand the definition of REBIND to ensure this (NORMAL designates the obvious predicate true just of normal-form s-expressions):

```
(DEFINE REBIND                                                    (S4-697)
   (LAMBDA EXPR [VARIABLE BINDING ENVIRONMENT]
      (COND [(EMPTY ENVIRONMENT) <the variable isn't bound>]
            [(= VARIABLE (1ST (1ST ENVIRONMENT)))
             (IF (NORMAL BINDING)
                 (RPLACN 2 ↑(1ST ENVIRONMENT) ↑BINDING)
                 (ERROR "Binding is not in normal form"))]
            [$T (REBIND VARIABLE BINDING (TAIL 1 ENVIRONMENT))])))
```

We turn next to the question of what environment is modified by a use of SET. In 3-LISP, when we have explicit access to any environment by reflecting, it is straightforward to write simple functions that modify arbitrary environments — such is the power of reflective code. But with regard to an object-level procedure such as SET, the only candidate environment that should be modified is the one in force at the point of processing of the call to SET. An instance of (SET <VAR> <TERM>), in other words, should "return" in a context in which the prior binding of <VAR> has been changed to the normal-form of <TERM>. The problem in 2-LISP is that we have no designator of this environment provided primitively; therefore in this dialect SET will have to be primitive, rather than defined in terms of REBIND (although we will retain REBIND, since we must use it, in certain cases, to modify bindings in closures).

Furthermore, when an extensional designator of a variable is desired (when, in other words, an equivalent of 1-LISP's SET is mandated), we will allow REBIND to be used with just two arguments; an absence of an explicit third argument, in other words, will default to

the environment currently in force. The semantic flatness of REBIND, however, will also be maintained; thus the following will fail:

```
(LET [[VARS ['X 'Y 'Z]]]                                        (S4-698)
   (MAP (LAMBDA EXPR [VAR] (REBIND VAR (+ 1 2)))
        VARS))
```

If it is intended to set X, Y, and Z to designate three, one would need instead to use

```
(LET [[VARS ['X 'Y 'Z]]]                                        (S4-699)
   (MAP (LAMBDA EXPR [VAR] (REBIND VAR ↑(+ 1 2)))
        VARS))
```

It should be clear that in talking in this way about modifying environments we are treading on rather unclear territory at the edge of 2-LISP, but not yet within the encompassing scope of 3-LISP. When we talk about SET modifying the "current" environment, then we speak about a change, in the meta-theoretic account, of the "environment" element of the ordered pair that represents the complete computational context (the other being the field). It is in order to accomodate just this kind of change that we have specified that environments are arguments to continuations — a facility in the meta-language that we have honoured but to date have not used. On the other hand, when we describe environment modifications in terms of structural field modifications to environment *designators* of the sort that play a role as ingredients in closures, we of course do not see any effect in the meta-theoretic environment term; we merely see a change in the field. We will make SET primitive in 2-LISP, and make evident its context modifications in the semantical account. We will also show how a semantic theory that constantly derives bindings from environment designators can be formulated (of the sort that would be required in a semantics for 3-LISP). What we leave open in our semantics of 2-LISP is the proper connection between the two — not because such connection could not be articulated, but rather because the connection is simply ugly. As we have said again and again, it is possible to construct correct semantical accounts of arbitrary behaviour, but that is not our purpose in doing semantics. Rather, we want semantical analysis to drive our design, and we already know that this environment question has its problems — problems that can only be solved in a reflective system. It would therefore not repay the investment to document this fact in a formal meta-language.

Three questions remain: a) what is the consequence of setting or rebinding a variable that is unbound? b) what is the status of the context in force during the processing of

expressions read in by the "READ-NORMALISE-PRINT" loop? and c) how does the environment encoded in a closure relate to the context in which the closure is constructed? In the most general sense, these all reduce to a single question: what is the *identity* of environment designators?

The answer is simple; we will explain it, and then review the motivation (it will be easier to defend the behaviour if the behaviour is made clear first). It is assumed that the environment encoded in a closure is, and remains, isomorphic to the meta-theoretic environment in force when the closure was constructed. There are two ways this can be thought about: either subsequent SETs will, as well as modifying the environment, also modify any encodings of that environment. More practically, one can assume that the environment is always driven off an encoding of an environment, and that SET merely engenders structural modifications to that encoding (this is of course the view that would be adopted by any implementation).

When a closure is constructed, the encoding of the current environment is provided to <EXPR> as its first argument. That encoding — a rail of two-element rails, as mentioned earlier — encodes the binding of some number of variables. When the closure is reduced with arguments, the body of the closure will be normalised in an environment consisting of the environment encoded in the closure *extended* with bindings of the closure's formal parameters, as appropriate. Suppose that a closure contained a pattern containing three formal parameters. Then the body of the closure will be normalised in an environment encoded in a rail whose first three elements will be two-element rails encoding the bindings particular to the given reduction, and whose third tail will be — will actually be structural identical with — the encoding found in the closure.

In the vast majority of cases, in other words, the entire set of environment designators throughout the system will form a tree, sharing a "tail" (the encoding of E_0), but otherwise branching out in a fashion the encodes the embedding structure of the program that has generated them.

Since all manner of consequences follows from this design decision, it is necessary to make it crystal clear. Suppose that in the initial environment (to be discussed below) we define a FACTORIAL procedure in the usual way; we would then have the following closure:

```
FACTORIAL      ⇒      (<EXPR> E0 '[N] '(IF ... ))                    (S4-700)
```

Suppose we were then to process the following fragment in E_0:

```
(LET [[X 3]]                                                          (S4-701)
   (FACTORIAL X))
```

First we expand the LET:

```
((LAMBDA EXPR [X]                                                     (S4-702)
    (FACTORIAL X))
 3)
```

By what has just been said, the body of this reduction — the term (FACTORIAL X) — will be processed in an environment E_1 in which X is bound to the numeral 3. This much has been clear for many sections; what we are currently making evident is that there will be constructed a *designator* of E_1 of exactly the following form: it will be a rail, whose first *element* will be:

```
['X '3]                                                               (S4-703)
```

and whose first *tail* will be E0.

Of what consequence is this? It can be noticed in two ways. First, suppose that we processed the following:

```
(LET [[X 3]]                                                          (S4-704)
   (LAMBDA EXPR [K] (+ K X)))
```

This would return a closure as follows:

```
(<EXPR> [['X '3] ... E0] '[K] '(+ K X))                               (S4-705)
```

where by " ... E0]" we mean simply that the first tail of the first argument to <EXPR> in this closure is E_0. However if instead we processed:

```
(LET [[X 3]]                                                          (S4-706)
   (LET [[TEMP (LAMBDA EXPR [K] (+ K X))]]
      (BLOCK (SET X 100)
             TEMP)))
```

we would be returned a closure as follows:

```
(<EXPR> [['X '100] ... E0] '[K] '(+ K X))                             (S4-707)
```

The reason, of course, is that the intervening SET modified the binding of X in the very environment encoded in the TEMP closure, and, as we said in the paragraph ealier, to modify the environment *is* to modify the environment designator.

Given this brief introduction from a behavioural point of view, we can turn back a little to motivation and explanation. First, consider the sentence ending the previous paragraph: "to modify the environment *is* to modify the environment designator". Strictly speaking, no sense can be made of the sentence "an environment was modified", given our ontology, for we have said that environments are abstract sequences of pairs of atoms and bindings, and abstract sequences, being mathematical entities, cannot be said to be *modified.* Therefore if SET cannot be strictly described as a procedure that modifies environments. Some other characterisation must serve instead.

Two possibilities present themselves. First, SET could simply be a procedure such that reductions in terms of it return in a context consisting of a *different* environment from that in which it was reduced. This would be a behaviour described in the following semantical equation:

$$\begin{aligned}&\Delta[E_0(\text{"}SET)] \equiv \lambda S.\lambda E.\lambda F.\lambda C. \\ &\qquad [\Sigma(NTH(2,S,F),E,F, \\ &\qquad\quad [\lambda\langle S_1,D_1,E_1,F_1\rangle\ . \\ &\qquad\qquad C(NTH(1,S,F_1),E_2,F_1)]] \\ &\quad \textit{where}\ [\ \forall A \in \textit{ATOMS}\ [\ E_2(A) = [\ \textit{if}\ [A = (NTH(1,S,F_1))] \\ &\qquad\qquad\qquad\qquad \textit{then}\ S_1\ \textit{else}\ E_1(A)\ ]]]\end{aligned} \qquad \text{(S4-708)}$$

Note that it is the internalisation of SET that is crucial in this discussion; we assume throughout the following general computational significance

$$\begin{aligned}&\Sigma(E_0(\text{"}SET)) \\ &\quad \equiv \lambda E.\lambda F.\lambda C\ . \\ &\qquad C(\text{"}\textit{(<IMPR> }\underline{E_0}\textit{ [VAR TERM] (SET VAR TERM))}, \\ &\qquad\ [\lambda\langle S_1,E_1,F_1\rangle\ .\ NTH(1,S_1,F_1)], \\ &\qquad\ E,F)\end{aligned} \qquad \text{(S4-709)}$$

since SET merely returns the name of the variable modified (this in part to prevent us becoming used to a SET that can be used both for effect and for a value: in 3-LISP SET redexes will have no designation at all, and will return no result).

The characterisation given in S4-708 and S4-709 is simple, and, although possibly efficient of implementation, it is not what we want (nor does it capture the behaviour indicated in S4-706 and S4-707). The problem has to do with the identity of the encoding of the environments in closures. By the account just given, the re-binding effected by SET would be visible only so long as the environment in force during the processing of the SET redex was used. A quick examination of the general computational significance of pairs, as set forth in S4-38, in conjunction with the internalisation of EXPR, as manifested in S4-626,

reveals that, in general, this will not be long. In particular, every *reduction* normalises its body in the environment that is recovered from the one encoded in the closure extended by the mandated bindings, not in the environment in which the redex is itself processed. This fact means that the effects of SET, if its full significance were spelled out in S4-708 and S4-709, would be constrained only to the two reduction boundaries on each side of it. It would be constrained, in other words, to its *static* context.

The trouble is that SET is useful primarily for more long-range effects. It is for this reason that it must be used with caution, but it is for this reason that it exists. Static contexts, especially in a tail-recursive dialect, can by and large be adequately handled with standard LAMBDA binding.

A particular example arises at what is known as the "top-level": the READ-NORAMLISE-PRINT loop with which the user interfaces with the processor. So-called *global* variables are one standard practice involving the potential for long-range effects, as are procedure definitions. Suppose, for example, that we discover that we have defined FACTORIAL in the intensionally iterative fashion illustrated in S4-572 and S4-573, but incorrectly, as follows:

```
(DEFINE FACTORIAL                                                    (S4-710)
   (LAMBDA EXPR [N]
      (FACTORIAL-HELPER 0 1 N)))

(DEFINE FACTORIAL-HELPER                              ; This has a bug
   (LAMBDA EXPR [COUNT ANSWER N]
      (IF (= COUNT N)
          ANSWER
          (FACTORIAL-HELPER (+ COUNT 1) (* COUNT ANSWER) N))))
```

The trouble is that this FACTORIAL will return 0 for any argument. But if this were not noticed, and a variety of other procedures were defined, it is natural to assume that one should merely "re-define" FACTORIAL-HELPER correctly (as in S4-573, for example). Suppose we typed this to the user interface:

```
> (DEFINE FACTORIAL-HELPER ... the correct version ... )             (S4-711)
> FACTORIAL-HELPER
```

We say in the previous section that DEFINE is a macro that expands to SET. On the account illustrated in the examples above, where this affects the environment designator structurally, then any procedure defined in terms of it will be re-defined. This is the natural behaviour. If, however, SET were merely of the consequence illustrated in S4-708 and S4-709, those closures would not be affected. There would, in fact, be no way in which *subsequent*

behaviour could affect any *prior* procedure definitions. Any procedures that used FACTORIAL would have to be re-defined. Furthermore, this would recurse, so that any procedures that used *them* would similarly have to be redefined. And so on and so forth, up to the transitive closure of acquaintance. Finally, any mutually-recursive procedures would have to be explicitly redefined within the scope of a single call to Z*. This is impractical in the extreme.

It is worth considering for a moment just what our suggestion comes to. As was the case when we talked about *modifying environments*, it is similarly vacuous, strictly speaking, to talk about *modifying procedures* or *modifying functions*. We have said that SET modifies *environment designators*; we have said as well that closures contain *environment designators* within them, in a manner such that tails are shared. When a SET — and by implication a DEFINE — is processed, those environment designators will in turn be affected. Thus the closures containing them will, strictly speaking, be different. Thus a name bound to such a closure will, in a sense, be bound to a different closure — it will certainly designate a different function. This, of course, is exactly what we want. We said that if SET had no further effect than modifying the current *theoretic* context, then perhaps all prior definitions would have to be re-done. However what the side-effect SET engenders *is exactly the same thing* — it merely does so with less work. For by modifying shared environment designators, it changes the functions designated by exactly the transitive closure of those procedures that use FACTORIAL, those that use procedures that use FACTORIAL, and so forth.

There is no escape, in sum, from the fact that by redefining a *given* procedure one may thereby *affect the full significance of a wide variety of others*. It has been remarked in other contexts that our beliefs "face the tribunal of experience *tout court*".[8] What we have seen is that there are two ways in which this corporate effect can be realised in a formal system. In one scheme the structures encoding the designation of the wide variety of procedures can be linked in the field; then a single change to that field will affect the total set of defintions. In the other, each procedure is kept isolated one from the next; the consequence is that in order to engender the correct behaviour, the complete set of designators will have to be modified *explicitly*. It reduces, in other words, to a question of whether the wide-spread effect should be explicitly or implicitly engendered; that the effect must be wide-spread is simply a matter of fact. Our choice has been with the implicit.

One reason we may adopt the implicit change protocol is that it need not cause us undue concern. In particular, it remains possible to protect a given closure from any such ill effects, should this be desired. We may, for example, construct a procedure called PROTECTING, to be used as follows:

```
(DEFINE SUM-OF-SQUARES                                          (S4-712)
   (PROTECTING [SQUARE +]
      (LAMBDA EXPR ARGS
         (+ . (MAP SQUARE ARGS)))))
```

SUM-OF-SQUARES merely returns the sum of the square of its arguments; thus we have:

```
(SUM-OF-SQUARES 2 4 6 8)      ⇒    120                          (S4-713)
```

However the PROTECTING ensures that no subsequent re-definition of SQUARE or + will modify the procedures used by SUM-OF-SQUARES. We would have, in particular, the following behaviour:

```
> (SUM-OF-SQUARES 2 4 6 8)                                      (S4-714)
> 120
> (+ . (MAP SQUARE [2 4 6 8]))
> 120
> (DEFINE + *)                          ; Redefine + to multiply
> +
> (+ . (MAP SQUARE [2 4 6 8]))          ; A public version of the
> 147456                                ; body does something quite new,
> (SUM-OF-SQUARES 2 4 6 8)              ; but the changed definitions
> 120                                   ; weren't seen by SUM-OF-SQUARES
```

PROTECTING is simply defined; it merely depends on the observation that a protected SUM-OF-SQUARES of the sort depicted in S4-714 can be defined as follows:

```
(DEFINE SUM-OF-SQUARES                                          (S4-715)
   (LET [[SQUARE SQUARE]
         [+ +]]
      (LAMBDA EXPR ARGS
         (+ . (MAP SQUARE ARGS)))))
```

This works because the embedded LAMBDA term is closed in an environment in which the atom SQUARE is bound to the binding that SQUARE had in the total surrounding environment; the closure thus retains its own private copy of that binding. Thus we can define PROTECTING as a macro so that expressions of the form

```
(PROTECTING [A1 A2 ... Ak] <BODY>)                              (S4-716)
```

expand to

```
(LET [[A1 A1] [A2 A2] ... [Ak Ak]]                                    (S4-717)
    <BODY>)
```

The appropriate definition of PROTECTING will be given in section 4.d.vii.

The procedure PROTECTING works only because DEFINE uses Z; this may not have been clear from just a cursory glance at the example. It is worth examining this in some detail, for we are at an excellent position to observe how our 2-LISP definition of DEFINE in terms of construction of circular closures differs noticeably from the practice normally employed to support recursive definitions. In particular, the behaviour we have adopted for SET implies that *if* the second argument to SET constructs a closure, then the binding effected by the SET will be visible from that closure. This fact is used by all standard LISP systems to support top-level recursive definitions; as we see in the following example, if we do not exercise it too strenuously it apparently yields the correct behaviour for recursive definitions in 2-LISP as well. As an example, we will assume that our primitive addition procedure accepts only two arguments, and will define a new procedure called ++ that will add any number of arguments. The definition will be recursive, but, rather than using DEFINE, we will for illustration merely use SET:

```
> (SET ++ (LAMBDA EXPR ARGS                                           (S4-718)
             (IF (EMPTY ARGS)
                 0
                 (+ (1ST ARGS) (++ . (REST ARGS))))))
> ++
```

In spite of the fact that we used SET rather than DEFINE for a recursive definition, it still approximately works, since the closure is constructed in the top-level environment, and the binding established by SET will be in that environment, visible to the procedure when it is reduced:

```
> (++ 1 2 3 4 5)                                                      (S4-719)
> 15
> (++)
> 0
```

In typical LISP systems DEFINE differs from SET because procedural definitions are not considered normal *values*, but other than this difference, immaterial in this discussion, the effect is the same. However using SET plus the global environment to implement recursion will fail to allow PROTECTING to work. Suppose for example we defined ++ as in S4-718 and then defined a SUM-OF-SQUARES as follows:

```
(DEFINE SUM-OF-SQUARES                                          (S4-720)
   (PROTECTING [++]
      (LAMBDA SIMPLE ARGS (++ . (MAP SQUARE ARGS)))))
```

This would work as long as ++ was not re-defined:

```
> (SUM-OF-SQUARES 1 2 3 4)                                      (S4-721)
> 30
> (SUM-OF-SQUARES 2 4 6 8)
> 120
```

However it would fail to protect ++, as the following illustrates. First we change ++ to multiply, rather than add, its arguments:

```
> (SET ++                                                       (S4-722)
       (LAMBDA SIMPLE ARGS                  ; Redefine ++ to be **
          (IF (EMPTY ARGS)
              1
              (* (1ST ARGS) (++ . (REST ARGS))))))
> ++
> (++ 1 2 3 4)
> 24
> (++)
> 1
```

By assumption SUM-OF-SQUARES was intended to be protected from this redefinition. However this is not the case; we in fact get quite an odd result, which is neither the expected SUM-OF-SQUARES nor the "product of squares" that *would* have resulted had the (PROTECT [++] ...) term been missing:

```
> (SUM-OF-SQUARES 1 2 3 4)                                      (S4-723)
> 677                              ; Should be 120; furthermore,
> (** . (MAP SQUARE [1 2 3 4]))    ; the "product" of squares
> 576                              ; is only 576.
```

What has happened is this: in the private binding of ++ that the definition of SUM-OF-SQUARES obtained in virtue of the form (PROTECTING [++] ...), ++ was bound not to a circular closure, but to a closure formed in the top-level ("global") environment. Thus the *internal use* of the name "++" within the recursive definition was affected by the re-definition of ++ in S4-722. What SUM-OF-SQUARES obtained, in other words, was only a *single* level of protection — not what was intended at all.

If, on the other hand, ++ had been defined using DEFINE rather than SET, as it ought to have been (the only difference between this and S4-718 is that DEFINE is used rather than SET):

```
> (DEFINE ++                                                    (S4-724)
    (LAMBDA EXPR ARGS
      (IF (EMPTY ARGS)
          0
          (+ (1ST ARGS) (++ . (REST ARGS))))))
> ++
```

then a completely protected SUM-OF-SQUARES would have been defined in S4-720. We would have, in other words (this is intended as a continuation of S4-724):

```
> (DEFINE SUM-OF-SQUARES                                        (S4-725)
    (PROTECTING [++]
      (LAMBDA SIMPLE ARGS (++ . (MAP SQUARE ARGS)))))
> SUM-OF-SQUARES
> (SUM-OF-SQUARES 2 4 6 8)
> 120
> (SET ++
       (LAMBDA SIMPLE ARGS                    ; Redefine ++ to be **
         (IF (EMPTY ARGS)
             1
             (* (1ST ARGS) (++ . (REST ARGS))))))
> ++
> (++ 1 2 3 4)
> 24
> (SUM-OF-SQUARES 2 4 6 8)                   ; SUM-OF-SQUARES is properly
> 120                                        ; protected.
```

This of course works because the closure to which SUM-OF-SQUARES obtains a private binding in turn contains its own private recursive access; it does not depend on the global availability of the name ++ within itself.

Some dialects, such as SEUS, have been proposed in which the ability to protect bindings within closures is provided as a primitive extension to the definition of LAMBDA. Once again we have seen that such functionality arises from the proper treatment of recursion, and from the discrimination between the public and internal recursive names of procedures. No additions are required to 2-LISP to support PROTECTING; furthermore, it behaves correctly not only because of the first level insight embodied in its definition in S4-717, but also because of the proper behaviour of Z.

It should be admitted in passing that protected procedures can of course always be redefined — the protection, in other words, can always be over-ridden — by obtaining explicit access to the enclosed environment designator. In particular, we could have (continuing S4-726):

```
> (REBIND '++ ↑** (ENV ↑SUM-OF-SQUARES))                        (S4-726)
> (<EXPR> ... )
> (SUM-OF-SQUARES 2 4 6 8)
```

```
> 147456
```

This example illustrates a tension — perhaps better seen as a dialectic — that we will encounter again and again in 3-LISP. Every attempt to detach one process (or program or structure) from the affects of another can be over-ridden by the second, if the latter avails itself of meta-structural and reflective powers. Similarly, the second process's attempts to over-ride the first process's intentions can likewise be over-ridden by the first process, by using yet more potent reflection. There is no way to do anything in 3-LISP that someone else cannot control and modify by rising one level higher than you rose. This has both its benefits and its troubles, as we will see.

One place that PROTECTING is useful is in the definition of DEFINE. The following code (MACROS in general will be discussed in section 4.d.v) is protected against subsequent re-binding of the atom Z:

```
(DEFINE DEFINE                                                    (S4-727)
   (PROTECTING [Z]
      (LAMBDA MACRO [LABEL FORM]
         `(SET ,LABEL (,↑Z (LAMBDA EXPR [,LABEL] ,FORM))))))
```

In general it has been our approach to consider semantics first, to define behaviour subsidiarily, and finally to give implementing mechanism a definite third place in order of importance. In the present instance, however, we have motivated and defended our design of SET on the basis of behaviour, for a simple reason: the import of SET is behavioural import; SET is not interesting in terms of its own designation. However in a sense our prevailing interest has remained, since the behaviour we have considered has to do with what functions *other* structures will ultimately designate, based on what effects SET unleashes on the field. What we wanted was, in a controlled way, to change the designation of previously-defined closures; what we have observed is that defining closures in terms of shared rails, coupled with an adequate fixed point procedure, yields a mechanism that supports this behaviour.

Given this design choice, it is natural to turn to the definition of the "top level" user interface. Because of two things, however, we will put this task off a little while yet. First, being essentially a behavioural matter, it is most easily explained with the aid of the meta-circular processor, which we examine in section 4.e. Second, this interface is another place — we are encountering more and more of them — where the intermediate status of

environments makes 2-LISP less than elegant. For this reason we will postpone such a discussion entirely until well into chapter 5, where READ-NORMALISE-PRINT will be defined as a straightforward user procedure. Environments and environment designators will be first-class entities in that dialect; with them the rest is straightforward.

There is, however, one additional issue to be tackled here. We have never given explicit attention to the question of errors, such as the use of a variable in a context in which it is not bound. We will continue to ignore them, but we have a problem with SET, since it is not an error, in standard practice, to SET a variable in a context where it is not bound. Indeed, in order to define *any* procedure with a previously unused name we use SET. The definition given in S4-691 of REBIND, on which SET is dependent, did not deal with unbound variables. If a variable is bound, it is clear that the effect of SET is at the point where it is bound, which has considerable consequences in terms of the public visibility of the change. It is easy simply to posit that SET (and REBIND) should *establish* a binding if there was none before. The question, however, is *where* in the environment structure such a binding should be inserted.

The only reasonable suggestions are at the "beginning" or at the "end" of the environment given to REBIND as its third argument — no other place is distinguished. Both pragmatics and analysis suggest the end — at the maximally visible place, in other words. That this is pragmatic is suggested by the following example:

```
(LET [[X (FACTORIAL 100)]                                    (S4-728)
      [Y (EXPONENTIAL 100)]]
    (IF (> X Y)
        (SET BIG-FUNCTION FACTORIAL)
        (SET BIG-FUNCTION EXPONENTIAL)))
```

Given that BIG-FUNCTION is not bound in the context of the body of the LET, a protocol selecting the *beginning* of the context's environment as a place to establish a new binding would mean that upon return from the LET in S4-728, the binding of BIG-FUNCTION would have been discarded, along with the bindings of X and Y. This would seem contrary to the apparent intention.

The "end" decision is at least suggested by analysis, as well. It would be possible to define the initial environment to contain bindings of *all* atoms: as we have said, a handful are bound to the primitively recognised closures; the rest could be bound to a distinguished and presumably non-designating structure, such as a special token <UNBOUND>. To do this

would complicate our dialect, since a new single-element structural category would have to be introduced. This token would fall outside of the range of any primitive function or procedure, so that any *use* of an unbound atom would engender an appropriate error. Then SET could be defined without regard to actually unbound atoms, since there would be none of them.

It only adds confusion to have a binding whose sole purpose is to encode the fact that an atom is not bound (this is reminiscent of an "end-of-file" token being used to indicate that a stream has been exhausted). Nonetheless, under this proposal all SETs to otherwise unbound atoms would be made visible to everyone — compatible with our suggestion that they add a binding at the end, rather than to the beginning, of the environment designator.

In sum, then, we will assume approximately the following definition of REBIND:

```
(DEFINE REBIND                                                    (S4-729)
   (LAMBDA EXPR [VAR BINDING ENV]
      (IF (NORMAL BINDING)
          (REBIND* VAR BINDING ENV)
          (ERROR "Binding is not i  normal form"))))

(DEFINE REBIND*                                                   (S4-730)
   (LAMBDA EXPR [VAR BINDING ENV]
      (COND [(EMPTY ENV) (RPLACT 0 ↑ENV ↑[[VAR BINDING]])]
            [(= VAR (1ST (1ST ENV)))
             (RPLACN 2 ↑(1ST ENV) ↑BINDING)]
            [$T (REBIND VAR BINDING (REST ENV))])))
```

The primitive use of SET, futhermore, and the use of REBIND with only two arguments, can be assumed to follow this protocol, with the appropriate environment designator provided automatically by the 2-LISP processor. This temporary inelegance will of course be dispensed with in the next chapter.

Finally, we need to discharge a debt we have carried for a long while: the use of $\underline{E_0}$ in primitive closures. We need to establish, in other words, the structure of the encoding of the initial environment. All of the ingredients to the answer have been set out; we need merely to assemble them. We said that there are thirty-two atoms bound to primitive closures, and there are no other privileged binds. Thus $\underline{E_0}$ (we have also called it <E0>) is (type-equivalent to) the following rail:

```
E0: [['TYPE      '(:X :E0 '[X] '(TYPE X))]                                    (S4-731)
     ['=         '(:X :E0 '[A B] '(= A B))]
     ['+         '(:X :E0 '[A B] '(+ A B))]
     ['-         '(:X :E0 '[A B] '(- A B))]
     ['*         '(:X :E0 '[A B] '(* A B))]
     ['/         '(:X :E0 '[A B] '(/ A B))]
     ['PCONS     '(:X :E0 '[A B] '(PCONS A B))]
     ['RCONS     '(:X :E0 'ARGS '(PCONS . ARGS))]
     ['SCONS     '(:X :E0 'ARGS '(SCONS . ARGS))]
     ['CAR       '(:X :E0 '[P] '(CAR P))]
     ['CDR       '(:X :E0 '[P] '(CDR P))]
     ['NTH       '(:X :E0 '[INDEX VECTOR] '(NTH INDEX VECTOR))]
     ['TAIL      '(:X :E0 '[INDEX VECTOR] '(TAIL INDEX VECTOR))]
     ['LENGTH    '(:X :E0 '[VECTOR] '(LENGTH VECTOR))]
     ['PREP      '(:X :E0 '[EL VECTOR] '(PREP EL VECTOR))]
     ['RPLACA    '(:X :E0 '[PAIR A] '(RPLACA PAIR A))]
     ['RPLACD    '(:X :E0 '[PAIR D] '(RPLACD PAIR D))]
     ['RPLACN    '(:X :E0 '[INDEX RAIL EL] '(RPLACN INDEX RAIL EL))]
     ['RPLACT    '(:X :E0 '[INDEX RAIL TAIL] '(RPLACT INDEX RAIL TAIL))]
     ['NAME      '(:X :E0 '[X] '(NAME X))]
     ['REFERENT  '(:X :E0 '[X] '(REFERENT X))]
     ['READ      '(:X :E0 '[] '(READ))]
     ['PRINT     '(:X :E0 '[S] '(PRINT S))]
     ['TERPRI    '(:X :E0 '[] '(TERPRI))]
     ['NORMALISE '(:X :E0 '[EXP] '(NORMALISE EXP))]
     ['REDUCE    '(:X :E0 '[PROC ARGS] '('REDUCE PROC ARGS))]
     ['EXPR      'X: (:X :E0 '[ENV PATTERN BODY] '(:X ENV PATTERN BODY))]
     ['IMPR      'I: (:X :E0 '[ENV PATTERN BODY] '(:I ENV PATTERN BODY))]
     ['MACRO     'M: (:X :E0 '[ENV PATTERN BODY] '(:M ENV PATTERN BODY))]
     ['LAMBDA    '(:I :E0 '[TYPE PAT BODY] '(↓TYPE (ENV) PAT BODY))]
     ['SET       '(:I :E0 '[VAR FORM]
                          '(REBIND VAR (NORMALISE FORM) (ENV)))]
     ['IF        '(:I :E0 '[PREM C1 C2]
                          '(IF (= 'ST (NORMALISE PREM))
                               (NORMALISE C1)
                               (NORMALISE C2)))]]
```

4.d. Meta-Structural Facilities

We turn next to a consideration of *meta-structural* questions and facilities: a particularly important step in the progression towards 3-LISP. We have already encountered a variety of practices having to do with the designation of elements of the field: all handles, for instance, are meta-structural in this sense. What we have not examined, however, are the protocols for "crossing levels", of which there are a variety of kinds. This section, however, will be comparatively brief, for two reasons. On the one hand those meta-structural capabilities that deal purely with the mentioning of uninterpreted structures are quite simple, and hence easily explained. The other primitives, on the other hand, like NORMALISE and REDUCE, that involve us in a shift of level of processing, are far from simple, but they will also be much better handled in 3-LISP. In the present section, therefore, we will examine such facilities in just enough detail to convince the reader that our development of 2-LISP should be abandoned, and that we should progress to a fully reflective dialect.

The section will proceed as follows: in 4.d.i we will look at NAME and REFERENT — the functions that have stood behind our ability to use "up" and "down" arrows ("↑" and "↓") from time to time in previous examples. The next two sub-sections examine NORMALISE and REDUCE; in 4.d.iv we will look at intensional procedures (IMPRS), and then in 4.d.v at macros and at the 2-LISP version of the so-called "backquote" notation. We will at that point have completely introduced the dialect; the final section, by way of review, will re-examine the "semantical flatness" that we promised to retain throughout the design of 2-LISP, and show how this property is true of all of 2-LISP, in spite of its meta-structural capabilities.

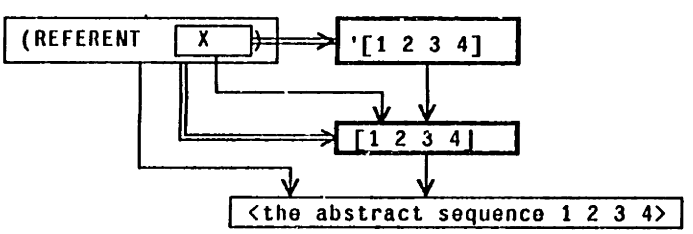
4.d.i. NAME and REFERENT

It was made clear in the previous chapter that normal-form designators normalise to themselves. It follows from this that there is no clear way to "strip the quote" off an expression. For example, suppose that some expression <EXP> designates a rail [1 2 3 4] — <EXP>, for example, might be (PREP '1 (RCONS '2 '3 '4)). We know that <EXP> would normalise to the handle '[1 2 3 4] — the normal-form designator of that rail. If we

wanted to bind a variable Y to that handle, we could use (LET [[Y <EXP>]] ...) as usual, since in that form <EXP> will be normalised prior to binding. On the other hand, if we should want to bind a variable Y to the expression that <EXP> *designates*, we cannot rely on any number of applications of the normalisation process, since normalisation is not a level-crossing operation. Some further mechanism is required.

In general, what we are looking for is a function that would map any expression <EXP> onto the entity designated by <EXP>. Such a function is not new to us, of course: it is the main semantical interpretation function Φ. In order to construct a composite expression in the syntactic domain we therefore need to be able to *designate the interpretation function* Φ: this is what we primitively require of the closure bound in the initial environment to the atom REFERENT. Thus, any application of the form (REFERENT <EXP>) is mandated to designate that entity designated by *the expression designated by <EXP>*, since REFERENT takes its argument in a normal, extensional position. This "double de-referencing" is entirely analogous to LISP's EVAL, which *doubly evaluates* its argument (1-LISP's (EVAL ''A) evaluates to A, not to 'A). While 2-LISP's main processor function NORMALISE is idempotent (Ψ = Ψ∘Ψ), the declarative interpretation function is not (Φ ≠ Φ∘Φ), just as 1-LISP's processor function was not (EVAL ≠ EVAL∘EVAL). Therefore Φ(⌜(*REFERENT EXP*)⌝) is different from Φ(EXP).

For example, consider the situation just described where <EXP> designates the rail [1 2 3 4]. Then the composite expression (REFERENT <EXP>) designates the designatum of that term designated by <EXP>, which is to say, (REFERENT <EXP>) designates the designatum of [1 2 3 4], which is the four-element *sequence* consisting of the first four positive integers. The situation is pictured in S4-735 (we assume that <EXP> in this case is the atom X):



(S4-735)

What then does the expression (REFERENT X) *normalise to*? It must normalise to the normal form designator of the sequence just mentioned, which is the rail [1 2 3 4]. Thus the claim that the atom REFERENT designates Φ, plus the normalisation mandate, yields directly that

REFERENT is the proper "quote-removing" function. In sum:

```
X                ⇒   '[1 2 3 4]                    (S4-736)
(REFERENT X)     ⇒   [1 2 3 4]
```

Note that REFERENT was defined purely *declaratively*: we did not define some new interpretive behavioural *procedure* called DE-REFERENCE to be executed when we want to get the designatum of some expression. It was entirely adequate (to say nothing of simpler) merely to give a primitive name to the semantical interpretation function: the procedural consequence of generating the referent, given a term, was supplied by the procedural consequence already embodied in the normalisation process. Put another way, although primitive functions have to be defined to *designate* to the semantical functions, no additional *behavioural* features need to be added to the interpreter: the standard processor is sufficient. This will prove true also when we refer to explicit normalisations — even including the function NORMALISE, which we will need to designate only *declaratively*, as will be seen in section 4.e.ii.

As mentioned in section 4.b.viii, we define a notational abbreviation for the reference function. In particular, notations of the form:

```
"↓" _ <notation>                                   (S4-737)
```

will be taken as abbreviatory for:

```
"(REFERENT" _ <notation> _ ")"                     (S4-738)
```

Thus we can write ↓X in place of (REFERENT X), for simplicity.

Some further examples:

```
↓'[1 2 3 4]                          ⇒   [1 2 3 4]          (S4-739)
↓(PREP '1 (RCONS '2 '3 '4))          ⇒   [1 2 3 4]
↓↓''''A                              ⇒   ''A
↓(+ 2 3)                             ⇒   <TYPE-ERROR:>
(+ . ↓(RCONS '1 '2))                 ⇒   3
(LET [[X '$F]] [(TYPE X) (TYPE ↓X)]) ⇒   ['BOOLEAN 'TRUTH-VALUE]
↓'(PCONS 'A 'B))                     ⇒   '(A . B)
↓(PCONS 'PCONS (RCONS ''A ''B))      ⇒   '(A . B)
↓(PCONS 'A 'B)                       ⇒   <ERROR: "A" is undefined>
```

The last three examples in this list indicate not only that two reference relationships play a role in the semantics of REFERENT (the one between the argument and its referent — since REFERENT is extensional — and the one between that designated expression and *its* referent, which is the mapping that REFERENT recovers), but two *normalisations* as well. To see why

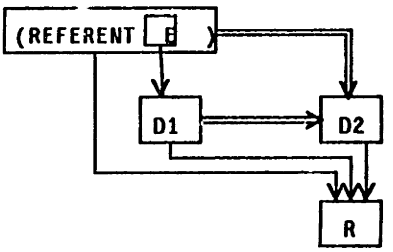
this must be so, we will look at two types of example. First, it is straightforward that if the formal argument in an application to REFERENT involves a side-effect, that side-effect will occur in the course of normalising the application, as the following session illustrates:

```
> (SET X '[1 2 30 4])                                          (S4-740)
> '[1 2 30 4]
> ↓(RPLACN 3 X '3)
> [1 2 3 4]
> ↓(BLOCK (PRINT 'DE-REFERENCING!) ''OK) DE-REFERENCING!
> 'OK
```

What is less obvious, however, is that a *second* normalisation is required in all applications in terms of this function:

```
> (SET X '[1 2 30 4])                                          (S4-741)
> '[1 2 30 4]
> ↓(PCONS 'RPLACN '[3 X '3])
> '[1 2 3 4]          ; The RPLACN happened as well as the PCONS
> X
> '[1 2 3 4]
```

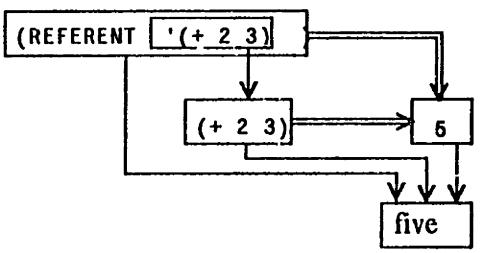
This double normalisation, it turns out, is mandated by the conjunction of standard computational considerations, and the declarative semantics of REFERENT, as the following diagram illustrates (as usual, single-tailed arrows represent designation (Φ), double-tailed arrows signify normalisation (Ψ), and heavily-outlined boxes surround expressions in normal-form). Given an expression of the form (REFERENT <E>), <E> is normalised as usual in order to determine its referent, which is called D1 in the figure. This is simply because REFERENT is declaratively an extensional function: procedurally an EXPR. Thus the whole term (REFERENT <E>) designates the *referent of* D1, which is called R in the figure. This much is not problematical, and, furthermore, is all there is to the declarative story. But the normalisation mandate requires more: given that (REFERENT <E>) designates R, then (REFERENT <E>) must *normalise to a normal-form designator of* R. The (or at least a) normal-form designator of R is the very expression *to which* D1 *normalises.* Furthermore, there is no tractable way of determining what D1 would normalise to without normalising it. Therefore D1 is normalised as well as <E>: the result of normalising D1, called D2 in the figure, is then returned as the result of normalising (REFERENT <E>).



(S4-742)

In terms of this figure, the example of S4-741 would be labelled as follows: E is the expression (PCONS 'RPLACN '[3 X '3]); the referent D1 of E is the expression (RPLACN 3 X '3); the referent R of D1 is the revised rail [1 2 3 4]. The normal-form designator of R is the handle '[1 2 3 4], which is the result of normalising D1: namely, D2.

Another simple example is illustrated in the following figure: the expression '(+ 2 3) designates the redex (+ 2 3); the referent of that redex is the abstract number five, of which the numeral 5 is the normal form designator. Therefore (REFERENT '(+ 2 3)) *designates* five, but *returns* 5:



(S4-743)

We have remarked in other contexts that although the extensionality of a function F implies that the *designation* of (FD . <ARGS>) will depend only on F and the designation of <ARGS> (where we assume that FD designates F), there are nonetheless *intensional* properties of the expression to which (FD . <ARGS>) reduces that may depend on the *intensional* form of <ARGS>. We have seen in the case of REFERENT redexes that the intensional dependencies can be rather complex: although (REFERENT <E>) *designates* the referent of the referent of <E>, the result of normalising (REFERENT <E>) may depend not only on the form of <E>, but also on the form (the intension) of the referent of <E>. This is what example S4-741 illustrated. In particular, D2, which is the result of normalising (REFERENT <E>), depended on the full computational significance not only of <E> (the expression (PCONS 'RPLACN '[3 X '3]) in the example), but also on the full computational significance of D1 ((RPLACN 3 X '3) in the example).

The second normalisation inherent in REFERENT will play a role in the reflective manoeuvring that comes into play in the next chapter. It will also come into focus when we discuss explicit calls to NORMALISE in the next section.

Although the discussion of REFERENT may seem straightforward, there is a considerable issue we have not yet discussed: the context used in the second normalisation engendered in the course of normalising a REFERENT redex. The answer is implicit in example S4-741, but needs to be made clear by stepping through some examples. First, suppose we normalise

```
(LET [[X 3]]                                                    (S4-744)
   (REFERENT 'X))          ⇒  3
```

It is natural that this should return the numeral 3, since 'X designates X, and X in this context designates three, and 3 is the normal-form designator of three. Indeed, this analysis is correct. Similarly, we might instead have the following:

```
(LET* [[X 3]                                                    (S4-746)
       [Y 'X]]
    ↓Y)                   ⇒  3
```

Again this will normalise to 3, as indicated. However the following is problematic:

```
(LET [[Y (LET [[X 3]] 'X)]]                                     (S4-746)
   ↓Y)                    ⇒  <ERROR: X is unbound>
```

The problem is that in the context in which the referent of Y is normalised, the variable X has no binding.

There are two levels at which we may react to this fact. On the one hand, the analysis, and this fact that results from it, seem natural enough. It is striking that by and large REFERENT is used in a situation where its argument expression designates a *normal-form* designator. From the fact that all normal-form designators are environment independent, it follows that they can be normalised in *any* context without error (indeed, they normalise to themselves). Furthermore, rather than *quoting* a designator from a context and passing it to another function as an argument, as the use of 'X in S4-746 exemplified, it is by and large more semantically defensible and practical to pass the *normal-form name* instead (constructed with NAME, discussed below). Thus, in place of S4-746, the appropriate and meaningful behaviour would have been this:

```
(LET [[Y (LET [[X 3]] ↑X)]]
   ↓Y)                          ⇒  3                          (S4-747)
```

This works because ↑X normalises to '3; thus Y is bound to '3. Hence (REFERENT Y) returns the numeral 3, because the referent of Y (the numeral 3) is a context-independent term.

From this point of view, then, we may simply observe these straightforward constraints. But there is something unsatisfying about such a shallow analysis. Though it is perfectly reasonable to point to a context-independent term and ask for its referent, it seems less reasonable to point to a *context-relative* term, and to ask for its referent (as we did, for example, in S4-746), without specifying what context we mean to use that term in. Suppose for example that I ask you for the referent of the proper noun phrase "*I Musici*", and you reply that it has no referent, because we are talking English. Sure enough *we* are talking English, but when I *mention* a term all bets are off, so to speak, on the relationship between the context in which you are intended to interpret the words I use to mention that term, *and the context you are intended to use to interpret the mentioned term.* It would be perfectly reasonable, for example, for me to ask you for the referent of "*I Musici*", taken as an Italian phrase. Similarly, I may ask you for the referent of the term "believe" for you, and ask you a moment later what its referent was *in pre-Rennaissance literature.*

It would be semantically preferable, in other words, if REFERENT were a function of *two* arguments, a term and a context. Thus (REFERENT <E> <C>) would designate the referent of the term designated by <E> in the context designated by <C>. It would *not* designate the referent of the term designated by <E> in the context being used to interpret the whole.

The problem with this suggestion, so far as 2-LISP goes, is that we have not in general provided facilities for passing environment designators as arguments. Such designators crept into closures, but it awaits 3-LISP before environment designators are fully integrated into the structure of the formalism. For the present, therefore, we will accept the simpler single-argument version of REFERENT, knowingly admitting that it is semantically improper. REFERENT has to do with crossing processes or interpreters — a subject beyond 2-LISP's ken.

It is useful to characterise REFERENT semantically, in part to illustrate with precision the points just made. First, without regard to context, we have the following simple

equations. Declaratively we expect this:

$$\Phi E_0(\text{"}REFERENT) = EXT(\Phi) \qquad \text{(S4-748)}$$

Procedurally, the situation is mildly more complex:

$$\Psi E_0(\text{"}REFERENT) = EXPR(\Psi \circ \Phi) \qquad \text{(S4-749)}$$

The proper treatment of full significance and context, however, demands a more complex story.

As is common with primitives, the full significance Σ of the primitive REFERENT closure is straightforward:

```
Σ[E0("REFERENT)]                                                    (S4-750)
  = λE.λF.λC
      C("(<EXPR> E0 '[TERM] '(REFERENT TERM)),
        [λS1,λE1,λF1 .
          Σ(S1,E1,F1,
            [λ<S2,D2,E2,F2> .
                Σ(NTH(1,D2,F2),E2,F2,[λ<S3,D3,E3,F3> . D3])])]
        E,F)
```

Intuitively, we expect REFERENT to designate a function that designates the referent (D_3) of the referent (D_2) of the single argument (S_1) with which it is called. Actually the story is a little more complex: D_2 is the *sequence* designated by REFERENT's argument, and D_3 is the referent of D_2's *single element*, because of our overall single-argument bent. Otherwise the simple story holds. As expected, the contexts yielded at each step of the way are passed through to the subsequent determinations of reference.

The internalised REFERENT function is also the straightforward consequence of the decisions just made. We have, in particular:

```
Δ[E0("REFERENT)]                                                    (S4-751)
   = λS1,E1,F1,C1 .
       Σ(S1,E1,F1,
         [λ<S2,D2,E2,F2> .
            Σ(HANDLE⁻¹(NTH(1,S2,F2)),E2,F2,
              [λ<S3,D3,E3,F3> . C1(S3,E3,F3)]))])
```

Note the assumption that $NTH(1,S_2,F_2)$ is a handle. This will be always be true, because it is assumed that $NTH(1,S_2,F_2)$ must *designate* a structure, and the semantical type theorem tells us that all normal-form structure designators are handles. S_2 is guaranteed to be in normal-form because it is the result of a normalisation; if it designates a sequence (which it must), it will be a recil of normal-form designators of that sequence's elements. Hence the

precondition will be met in all cases in which REFERENT applies.

In contrast, we present the semantical equations governing the suggested two-argument REFERENT, where the second argument designates the environment (the field — the other part of the context — is as usual passed through by default). We will call it REFERENT_2. First we give the full significance (the portion that differs from S4-760 is underlined):

$$\Sigma[E_0(\text{"}\textit{REFERENT}_2)] \qquad \text{(S4-762)}$$
$$= \lambda E.\lambda F.\lambda C$$
$$C(\text{"}(\langle EXPR\rangle\ \underline{E_0}\ \text{'}[TERM]\ \text{'}(REFERENT\ TERM)),$$
$$[\lambda S_1,\lambda E_1,\lambda F_1\ .$$
$$\Sigma(S_1,E_1,F_1,$$
$$[\lambda\langle S_2,D_2,E_2,F_2\rangle\ .$$
$$\Sigma(\text{NTH}(1,D_2,F_2),\underline{\text{NTH}(2,D_2,F_2)},F_2,[\lambda\langle S_3,D_3,E_3,F_3\rangle\ .\ D_3])])]$$
$$E,F)$$

Note the use of $\text{NTH}(2,D_2,F_2)$ in the *second* argument position to the embedded call to Σ (in place of E_2): we of course assume that Φ is the appropriate interpretation function to yield environments from environment designators.

Slightly more problematic is the internalised function signified by REFERENT_2. A first attempt is this:

$$\Delta[E_0(\text{"}\textit{REFERENT}_2)] \qquad \text{(S4-763)}$$
$$= \lambda S_1,E_1,F_1,C_1\ .$$
$$\Sigma(S_1,E_1,F_1,$$
$$[\lambda\langle S_2,D_2,E_2,F_2\rangle\ .$$
$$\Sigma(\text{HANDLE}^{-1}(\text{NTH}(1,S_2,F_2)),\underline{\text{NTH}(2,D_2,F_2)},F_2,$$
$$[\lambda\langle S_3,D_3,E_3,F_3\rangle\ .\ C_1(S_3,E_2,F_3)])])$$

In general it is of course illegal, in the specification of an internalised function, to make substantive use of the *designations* returned as the second coordinate of embedded calls to Σ. We have violated this with respect to the environment argument because, as we have made clear, environments are theoretical entities *of the meta-theory*; thus Σ has paradigmatic rights to actual environments, rather than to environment designators. Note as well that we needed a slightly different final continuation; C_1 is given E_2, and E_3 is discarded. 3-LISP's more adequate treatment of environments will correct this lack.

It should not be surprising that REFERENT must be primitive: there is no other way, for example, in which the referent of a handle may be obtained, even though we said in section 4.a that the "HANDLE" relationship was one of bi-directional local accessibility (it is REFERENT that functionally embodies that locality aspect of the field). What is less clear is

whether *other* meta-structural capabilities — such as those provided by IMPRS, for example — redundantly provide this power. This is not the case, because REFERENT *crosses levels* in a way that no other functionality can.

The situation regarding naming of entities is in many ways analogous to that of referring to their referents, although it is somewhat simpler, and we have used the primitive NAME function more in previous examples. The task is the inverse of the one just considered: given a term designating some entity, what expression enables one to refer to a *normal-form designator* of that entity. For example, suppose we have a variable X that designates some number. If we normalise X we know that we will obtain a numeral that we can use; the question is how can we *mention* that numeral.

It should made clear straight away that the question is not the simpler one of merely being able to mention *any* designator of that entity, for this is trivial: one merely uses the appropriate handle. In particular, given any term <X> designating entity D, the term '<X> designates *one* designator of D. For example '(+ X Y) is guaranteed to designate a term that designates the referent of (+ X Y). What must also be provided, however, is the ability to mention a *context-independent, side-effect free, stable* designator; and this, it turns out, requires primitive support.

In this situation we require a primitive closure that designates the inverse designation function: that function that takes each entity in the semantical domain into (one of) its normal-form designator(s). We call this function NAME (although note that it designates not just any name, but a *normal-form* name of its argument). In a manner parallel to REFERENT, we have a notational abbreviation: expressions of the form:

```
"↑" _ <notation>                                        (S4-754)
```

are considered abbreviations for:

```
"(NAME" _ <notation> _ ")"                              (S4-755)
```

Again like REFERENT, NAME is defined purely declaratively, but from that definition the following examples follow directly:

```
↑$T                          ⇒   '$T                    (S4-756)
↑(= 3 4)                     ⇒   '$F
↑7                           ⇒   '7
↑(+ 3 4)                     ⇒   '7
↑(PCONS 'A 'B)               ⇒   ''(A . B)
(LET [[X 3][Y 4]] ↑(+ X Y))  ⇒   '7
```

```
↑↑↑↑(= 3 4)                    ⇒  ''''$F
```

Another set of examples makes clear the difference between applications in terms of the NAME function and corresponding handles:

```
(+ 2 3)                        ⇒  5                              (S4-757)
'(+ 2 3)                       ⇒  '(+ 2 3)
↑(+ 2 3)                       ⇒  '5
(TYPE (= 'A 'A))               ⇒  'TRUTH-VALUE
(TYPE '(= 'A 'A))              ⇒  'PAIR
(TYPE ↑(= 'A 'A))              ⇒  'BOOLEAN
(TYPE ↑'(= 'A 'A))             ⇒  'HANDLE
(TYPE '↑(= 'A 'A))             ⇒  'PAIR
'(TYPE (= 'A 'A))              ⇒  '(TYPE (= 'A 'A))
↑(TYPE (= 'A 'A))              ⇒  ''TRUTH-VALUE
```

From the properties of NAME a few corrollaries are provable. First, since expressions of the form ↑<EXP> always designate a *designator* of the referent of <EXP>, it is provable that they always designate an element of the structural field (all designators being structural field elements). Thus (TYPE ↑<EXP>) will always designate one of ATOM, PAIR, RAIL, HANDLE, BOOLEAN, or NUMERAL. In addition, expressions of the form ↑<EXP> will always *normalise* to handles, since all structural field elements' normal-form designators are handles.

All of these follow from the semantical equations governing NAME:

$$\Sigma[E_0(\text{"}\textit{NAME}\text{)}] \qquad \text{(S4-768)}$$
$$= \lambda E.\lambda F.\lambda C$$
$$C(\text{"}\textit{(<EXPR>}\ \underline{E_0}\ \textit{'[TERM] '(NAME TERM))},$$
$$[\lambda S_1,\lambda E_1,\lambda F_1\ .\ \Sigma(S_1,E_1,F_1,[\lambda\langle S_2,D_2,E_2,F_2\rangle\ .\ \mathrm{NTH}(1,S_2,F_2)])]$$
$$E,F)$$

$$\Delta[E_0(\text{"}\textit{NAME}\text{)}] \qquad \text{(S4-769)}$$
$$= \lambda S_1,E_1,F_1,C_1\ .$$
$$\Sigma(S_1,E_1,F_1,$$
$$[\lambda\langle S_2,D_2,E_2,F_2\rangle\ .\ C_1(\mathrm{HANDLE}(\mathrm{NTH}(1,S_2,F_2)),E_2,F_2)])$$

Note in S4-768 that the NAME closure designates an extensional function that maps terms onto *what they normalise to.* Thus NAME is not in fact strictly extensional, in spite of the fact that it is an EXPR (it is the only exception to the rule that procedural EXPRS are declaratively extensional, just as IF is the exception to the rule that procedural IMPRS are declaratively intensional).

NAME and REFERENT, being inverse functions, can be composed with rather interesting results. Thus, in general, "down-up" signifies (procedurally and declaratively) the identity function:

$$\forall S \in \mathcal{S}\ [\ \Sigma(\ulcorner\uparrow\downarrow\uparrow\underline{S}\urcorner) = \Sigma(S)\ ] \qquad \text{(S4-760)}$$

Of much more interest, however, is the other combination: what we call "up-down", referring to "↑↓E", the abbreviation for (NAME (REFERENT E)). A term of the form ↑↓E may fail to be extensionally equivalent to E: first, NAME is not strictly a *function*: some forms (such as functions) have more than one normal-form designator. Secondly, REFERENT is a partial function of the semantical domain; ↓EXP is defined only when EXP designates a term (an element of the structural field *S*); ↑↓JOHN will typically be ill-formed (assuming the atom JOHN designates a person John), since John the person is not a term. Similarly, ↑↓LAMBDA is semantically ill-formed, because the function that the atom "LAMBDA" designates is not a sign. In spite of these limitations, however, in section 4.e.iv we will prove a striking theorem: ↑↓<EXP> is always entirely equivalent, both procedurally and declaratively, to (NORMALISE <EXP>) (NORMALISE, of course, is *declaratively* the identity function: this merely states that NORMALISE — 2-LISP's Ψ — is designation-preserving: our main semantical mandate). It is for this reason that NORMALISE in 2-LISP need not be primitive (or, alternatively, NAME need not be — what we really prove is that they are interdefinable).

This will be pursued in greater depth in section 4.d.iv. Before leaving the NAME procedure, however, we have two final comments. First, no issues arise in connection with NAME of the sort that attend REFERENT, having to do with a second context and a second processing. As the semantical equations governing NAME demonstrate, only the single processing step common to all EXPRS is engendered by a NAME redex.

Secondly, for those familiar with his work, we should arrest any tendency to equate the 2-LISP NAME function with the operator that Richard Montague uses in his intensional logics[9] to designate the *intension* of a term (he also uses a prefix up-arrow). We have admitted that we have not reified intensions; therefore we provide no way, given a term X, to construct another term Y such that the *referent* of Y is the *intension* of X. Had we an adequate theory of intensionality, such a primitive would be useful. For the time being, however, our "↑" remains a simple meta-structural primitive, rather than a "meta-intensional" one. We use "↑X", in other words, to refer to the *name of the normal form* of term X; Montague uses "↑X" to refer to the *intension* of term X.

4.d.ii. NORMALISE and REDUCE

We are now in a position to examine explicit "calls" to the processor itself — forms, for example, like (NORMALISE '(CAR X)). Two procedures in particular will be provided. For historical compatibility we will call them "NORMALISE" and "REDUCE", although "NORMAL-FORM" would be more appropriate than "NORMALISE", and "REDUCTION" than "REDUCE", since they do not actually denote the interpretive *process per se*, but merely the function computed by that process.

Intuitively, there is no real difficulty with these procedures, once we recognise that they are standard extensional functions. In particular, (NORMALISE <A>) will designate <B> just in case the structure designated by <A> would normalise to <B>. Hence (NORMALISE <A>) will *normalise* to a normal-form designator of <B>. <A> must of course designate an expression (normalisation — Ψ — is only defined over S), and <B> will be an expression. Hence (NORMALISE <A>) will *return* a normal-form expression designator — a handle.

We observed in connection with REFERENT redexes that two normalisations were involved; the same is true with respect to NORMALISE, for much more obvious reasons. Since NORMALISE is an EXPR, the argument in a NORMALISE redex will be normalised; then, the expression that that expression designates will in turn be normalised. Some simple examples:

```
(NORMALISE ''[THIS IS A RAIL])          ⇒   ''[THIS IS A RAIL]      (S4-765)
(NORMALISE ''$T)                        ⇒   ''$T
(NORMALISE '$T)                         ⇒   '$T
(NORMALISE $T)                          ⇒   <ERROR>
(NORMALISE '3)                          ⇒   '3
(NORMALISE '(+ 1 2))                    ⇒   '3
(NORMALISE '(CAR '(A B C)))             ⇒   ''A
(NORMALISE (XCONS 'CDR ''(1 . 2)))      ⇒   ''2
```

Perhaps the easiest way to think about these examples is this: if you understand the argument to NORMALISE as *designating* an expression that appeared on the left side of our standard "⇒" arrow, what expression would appear on the right? Then the handle designating that right hand side expression is the result returned by the NORMALISE redex.

NORMALISE is of course idempotent; thus (NORMALISE (NORMALISE <X>)) will always have the same full significance as the simpler (NORMALISE <X>). Some examples:

```
(NORMALISE (NORMALISE '$F))                    ⇒   '$F              (S4-766)
(NORMALISE (NORMALISE '(+ 2 3)))               ⇒   '5

(LET [[X 1]]
   (BLOCK (NORMALISE '(SET X (+ X 1)))
          X))                                  ⇒   '2

(LET [[X 1]]
   (BLOCK (NORMALISE
             (NORMALISE '(SET X (+ X 1))))
          X))                                  ⇒   '2       ; Not '3
```

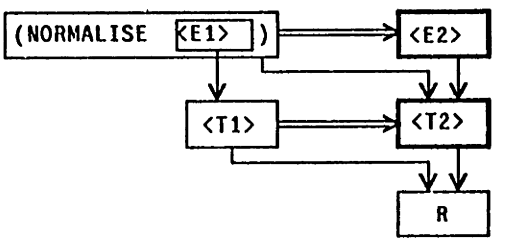
However this does not imply that normalising an explicit call to NORMALISE is indistinguishable from simply normalising an expression directly; whereas two *uses* of NORMALISE come to the same thing as a single *use*, *mentioning* NORMALISE is of course quite different from simply using it:

```
(NORMALISE '(= 3 4))                 ⇒   '$F      ; The first two    (S4-767)
(NORMALISE (NORMALISE '(= 3 4)))     ⇒   '$F      ; are equivalent, but
(NORMALISE '(NORMALISE '(= 3 4)))    ⇒   ''$F     ; the third is different.
```

The crucial fact about NORMALISE redexes is this: *they do not cross semantic levels.* Rather, they can be understood as if they *reach* down one level, but remain at that higher level looking down. In other words, whereas the semantic level of (NAME <E1>) is one level higher than the level of <E1>, and the semantic level of (REFERENT <E2>) is one level below that of <E2>, the semantic level of (NORMALISE <E3>) is the *same* as that of <E3>. This should come as no surprise: the salient fact about normalisation *in general*, as opposed to evaluation, is that is preserves semantic level; it is to be expected that explicit references to this function will themselves preserve semantic level.

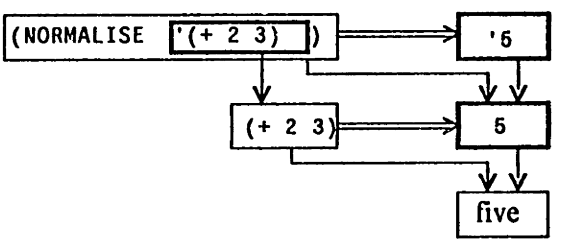
The general structure of the Ψ and Φ relationships among the constituents and results in a NORMALISE redex are shown in the following diagram. The idea is this: in general, given a redex of the form (NORMALISE <E1>), the argument <E1> will designate some term <T1>, which in turn presumably has some referent <R>. If <T1> were normalised, it would yield some other term <T2> that also designated <R>, but that was in normal form; this is what it *is* to normalise. Therefore (NORMALISE <E1>) *designates* T2. What then should (NORMALISE <E1>) *normalise* to? Clearly, to the normal-form designator of <T2>: an expression we have called <E2> in the diagram.

(S4-768)



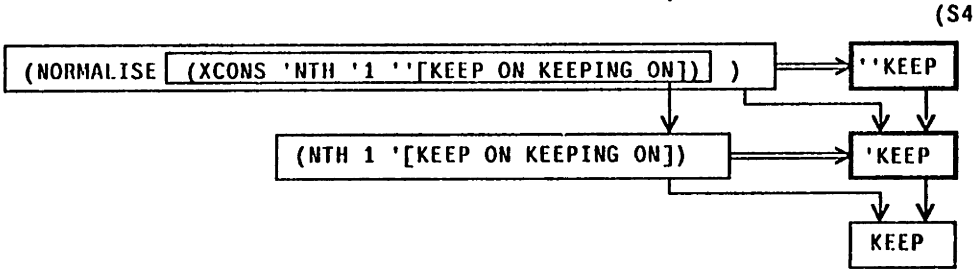
An simple example is pictured in the following diagram. <E1> is the handle '(+ 2 3); hence <T1> is the addition redex (+ 2 3), which designates an <R> of five. <T2>, therefore, the normal-form designator of five, is the numeral 5. Hence <E2>, the normal form designator of that numeral, is the handle '5.

(S4-769)



Finally, a slightly more complex case. In the following, <E1> is not in normal-form; it is the XCONS redex (XCONS 'NTH '1 ''[KEEP ON KEEPING ON]). Thus <E1> designates a <T1> that is the NTH redex (NTH 1 '[KEEP ON KEEPING ON]), which in turn designates an <R> that is the atom KEEP. The normal-form designator of this atom — the example's <T2> — is the handle 'KEEP. Hence <E2>, the normal-form designator of this handle, is the further handle ''KEEP.

(S4-770)



Given this much of an analysis, it is straightforward to present the formal semantics of NORMALISE. Without regard to the complexities of context and full signficance, we of course are aiming at the following:

$$\Phi(E_0(\text{"}NORMALISE)) = EXT(\Psi) \qquad \text{(S4-771)}$$

(This should be constrasted with S4-748's claim that $\Phi(E_0(\text{"}REFERENT)) = EXT(\Phi)$.) Procedurally, we will approximate this:

$$\Psi(E_0(\text{"}NORMALISE)) = EXPR(\Phi^{-1}\circ\Psi\circ\Phi) \qquad \text{(S4-772)}$$

Though Φ^{-1} is not in general well-defined (since Φ is many-to-one, Φ^{-1} is not in general a function), it happens that Φ^{-1} *is* well-defined over the range of Ψ, namely S. In other words we essentially have the following:

$$\forall S \in S\ [\Phi^{-1}(S) = HANDLE(S)] \qquad \text{(S4-773)}$$

Hence S4-772 reduces to

$$\Psi(E_0(\text{"}NORMALISE)) = EXPR(HANDLE\circ\Psi\circ\Phi) \qquad \text{(S4-774)}$$

More fully, however, we have the following full significance:

```
Σ[E0("NORMALISE)]                                                    (S4-775)
   = λE.λF.λC
       C("(<EXPR> E0 '[TERM] (NORMALISE TERM)),
         [λS1,E1,F1 .
           Σ(S1,E1,F1
             [λ<S2,D2,E2,F2> .
                 Σ(NTH(1,D2,F2),E2,F2,[λ<S3,D3,E3,F3> . S3])])],
         E,F)
```

and internalised function:

```
Δ[E0("NORMALISE)]                                                    (S4-776)
   = λS1,E1,F1,C1
       Σ(S1,E1,F1,
         [λ<S2,D2,E2,F2> .
            Σ(HANDLE⁻¹(NTH(1,S2,F2)),E2,F2,
              [λ<S3,D3,E3,F3> . C1(HANDLE(S3),E3,F3)])])
```

The underlined parts of these two equations highlight the only places in which they differ from the semantics of REFERENT, with which they should be compared. This fact — mandated by the meaning of the words, not something we have aimed for explicitly — begins to hint at the close relationship among NAME, REFERENT, and NORMALISE that will be brought to the fore in the "up-down" theorem of section 4.d.iv.

What these equations, and the examples presented earlier, make clear is that the second normalisation mandated by a NORMALISE redex happens in the context resulting from the processing of the NORMALISE arguments. All of the discussion as to why this is inelegant holds equally with respect to NORMALISE; this function should not be given just a single

argument; it should be given a context as well. We need not belabour this point here, because we will shortly begin to look at better ways of doing this. The meta-circular processor in section 4.d.iii will define a version of NORMALISE that takes not only an environment but a continuation argument; similarly the fully reflective NORMALISE of 3-LISP will be defined in terms of these same three arguments. Thus we will ultimately support such code as

```
(LET [[X '(+ X Y)]]                                              (S4-777)
   (NORMALISE X
              [['X '3]['Y '4] ... ]
              <CONT>))                    ⇒    '7
```

in which the use of X is relative to a different environment than the mention of X. The present inadequate single-argument version is merely intended to illustrate the kind of behaviour that the explicit use of NORMALISE can engender.

The situation regarding function application, and redex reduction — and therefore any explicit use of the REDUCE function — is entirely analogous to that regarding the general normalisation of expressions. The arguments about ultimately requiring a different context hold, but we will restrain our attention to the single-context version for the time being. It should be noted as well that we are defining a *reduction*, not an *application* procedure: as set forth in section 3.f.i, a correct definition of an APPLY procedure is both trivial and useless.

REDUCE is provided as much for convenience as necessity. We have, in particular, the following sorts of behaviour:

```
(REDUCE '= '[3 3])                          ⇒    'ST          (S4-778)
(REDUCE 'NTH '[1 '[BE BRIEF]])              ⇒    'BE
(REDUCE
   (NTH 2 [+ IF LAMBDA])
   (TAIL 1 '[(= 3 3) (= 3 4) 'YES 'NO]))    ⇒    ''NO
(REDUCE 'NORMALISE '['(+ 2 3)])             ⇒    ''5
```

REDUCE could have been defined as follows:

```
(DEFINE REDUCE                                                   (S4-779)
   (LAMBDA EXPR [PROCEDURE ARGS]
      (NORMALISE (PCONS PROCEDURE ARGS))))
```

We needn't, therefore, take the time to examine its semantics: they are a simple combination of the semantics of NORMALISE (presented in S4-775 and S4-776) together with the semantics of pairs, as given in S4-38.

Of more interest is a comparison between 2-LISP's REDUCE and 1-LISP's APPLY, in particular since we set out to define a dialect that would subsume at the object level all of the inessential reasons that APPLY was used in 1-LISP. This is particularly salient given the definition just presented in S4-779, since 1-LISP's APPLY *cannot* be defined as follows:

```
(DEFINE APPLY                          ; This is 1-LISP.          (S4-780)
   (LAMBDA (FUN ARGS)                  ; and it is also
      (EVAL (CONS FUN ARGS))))         ; incorrect.
```

We will look, therefore, at five diffierent examples using APPLY and REDUCE, in order to bring out the differences.

Consider first a case in which a perfectly ordinary redex would serve: as for example in 1-LISP's (CONS 'A 'B) and 2-LISP's corresponding (PCONS 'A 'B). If these forms were *designated* by a simple quoted form, they could be given as a single argument to each's dialect's name for its Ψ:

```
(EVAL '(CONS 'A 'B))              →     (A . B)      ; 1-LISP      (S4-781)
(NORMALISE '(PCONS 'A 'B))        ⇒   ''(A . B)      ; 2-LISP
```

The (double) lack of semantic flatness on 1-LISP's part is of course evident here, but otherwise the situations are not dissimilar. Furthermore, we can take the expression apart into "function" and "argument" components, and use APPLY/REDUCE, leading again to approximately similar constructs:

```
(APPLY 'CONS (LIST 'A 'B))        →     (A . B)      ; 1-LISP      (S4-782)
(REDUCE 'PCONS '['A 'B])          ⇒   ''(A . B)      ; 2-LISP
(REDUCE 'PCONS (RCONS 'A 'B))     ⇒   ''(A . B)      ; 2-LISP
```

Again there is no striking difference except the de-referencing behaviour of EVAL. In the first 2-LISP form (the second line of S4-782) we simply used explicit rail brackets to designate a rail, although the second 2-LISP form (the third line) was semantically equivalent, and more similar to the 1-LISP counterpart.

If we were to go no further, it might look as if the dialects were therefore moderately alike in these respects, but this is of course far from the case. For one thing, there is an unclarity, in the first line of S4-782, as to whether the second argument to APPLY *objectifies* the arguments (i.e., is a single designator of a *sequence* of arguments), or whether it *designates the appropriate argument expression* for the procedure in question. It should be clear that the 2-LISP REDUCE redexes (the second and third line) are firmly entrenched in the second of these two options, since REDUCE is throughout *meta-structural.* Any attempt to

use the first strategy would lead to an error:

```
(REDUCE 'PCONS ['A 'B])          ⇒   <ERROR: exp'd an s-expr>      (S4-783)
(REDUCE 'PCONS (SCONS 'A 'B))    ⇒   <ERROR: exp'd an s-expr>
```

although both of the following are perfectly acceptable:

```
(PCONS . ['A 'B])                ⇒   '(A . B)       ; 2-LISP       (S4-784)
(PCONS . (SCONS 'A 'B))          ⇒   '(A . B)       ; 2-LISP
```

Therefore we realise that the use of APPLY in S4-782 is really a case where the arguments have been *objectified*, rather than being a case where the argument expressions have been meta-structurally *designated*. In moving from a standard issue redex to one appropriate for APPLY, in other words, we were forced to give as APPLY's first argument a *designator of the procedure name*, but to give as APPLY's second argument a *designator of the sequence of argument values*, not a designator of the argument expression. 1-LISP's APPLY, as we can now see, is in terms of diagram S3-178 approximately a function from *function designators* (FD) and *arguments themselves* (A) onto either value designators or values, depending on whether the values themselves are structural entities.

That this is indeed the case is more clearly revealed in the next set of examples, where we consider a second type of circumstance, where rather than converting an expression that worked properly on its own, we actually consider a situation in which we need to use APPLY. In particular, if we let X designate a list of two atoms in the only way 1-LISP provides for doing that, and if we want to construct the pair consisting of these two atoms, then we must subsequently use APPLY:

```
(LET ((X (LIST 'A 'B)))                                            (S4-785)
   (APPLY 'PCONS X))             →    (A . B)       ; 1-LISP
```

On the other hand, we do *not* need to use REDUCE in 2-LISP:

```
(LET [[X ['A 'B]]] (PCONS . X))  ⇒   '(A . B)       ; 2-LISP       (S4-786)
```

Furthermore, it generates an error if we do, unless we explicitly extract the appropriate designator *of that designator of a sequence of atoms*:

```
(LET [[X ['A 'B]]]                                                 (S4-787)
   (REDUCE 'PCONS X))            ⇒   <ERROR: Expected an s-expression>
(LET [[X ['A 'B]]]
   (REDUCE 'PCONS ↑X))           ⇒   ''(A . B)      ; 2-LISP
```

Conclusion number one, therefore, is this: whereas APPLY is indicated in 1-LISP for argument objectification, that can be accomplished in 2-LISP by using non-rail CDRs. The second argument to REDUCE must designate an *argument expression*, not an objectified argument sequence, since REDUCE, unlike APPLY, is consistently meta-structural.

A third case, where APPLY is indicated in 1-LISP, arises when, informally, the "function" is the value of a term, rather than being the term itself. Now of course functions aren't terms: what is meant is that the term designates the function *name*. Since this *differs* from objectifying the arguments, standard LISPs typically have an APPLY variant to treat it, called APPLY* in INTERLISP and FUNCALL in MACLISP (we will use the INTERLISP terminology). Some examples:

```
(APPLY* '+ 2 3)                  →   5              ; 1-LISP      (S4-788)
(LET ((X 'CONS))
    (APPLY* X 'A 'B))            →   (A . B)        ; 1-LISP
```

However it is of course true in a higher-order dialect that no resort to explicit processor primitives is indicated in such a circumstance:

```
(LET [[X PCONS]] (X 'A 'B))      ⇒   '(A . B)       ; 2-LISP      (S4-789)
```

It must be admitted, however, that in the 1-LISP examples (S4-788) X is bound to a *designator of the constructor's name*; if we were to do the same in 2-LISP (that being a meta-structural operation) we too would either have to de-reference it before using it, or else would need to use REDUCE explicitly (but in the latter case we would have to designate the argument expression as well):

```
(LET [[X 'PCONS]] (X 'A 'B))     ⇒   <ERROR: Expd a function>   (S4-790)
(LET [[X 'PCONS]] (↓X 'A 'B))    ⇒   '(A . B)       ; 2-LISP
(LET [[X 'PCONS]]
    (REDUCE X '['A 'B]))         ⇒   ''(A . B)      ; 2-LISP
```

This is the circumstance regarding MAPS, as well (the "function" argument to MAP must be quoted in 1-LISP but not in SCHEME and 2-LISP), relating to the use of static versus dynamic scoping, and so forth. Once again, especially from the fact that the arguments to APPLY* (after the first one, which is the function) appear in *exactly* the same form as if the function's name were used explicitly in the first position of the redex, we can conclude that this use of a member of the APPLY family has to do with context-relative procedure specification, rather than with anything inherently meta-structural or objectifying of the processor.

This is made even clearer by considering a fourth situation in which one *does* have a designator of the appropriate argument expressions. Strikingly, in that case neither of 1-LISP's APPLY *or* APPLY* can be used; one must resort to EVAL. Suppose in the following examples that X is bound to 3 and Y to 4. In 1-LISP we have:

```
(LET ((A 'X) (B 'Y))                                                    (S4-791)
    (APPLY* '+ A B))          →   <ERROR: X not a number [sic]>
(LET ((C '(X Y)))
    (APPLY '+ C))             →   <ERROR: X not a number [sic]>
```

What one must resort to instead is this:

```
(LET ((A 'X) (B 'Y))                                                    (S4-792)
    (EVAL (CONS '+ (LIST A B)))  →   7          ; 1-LISP
(LET ((C '(X Y)))
    (EVAL (CONS '+ C))           →   7          ; 1-LISP
```

This is because APPLY, although *it itself* evaluates its arguments, doesn't *re-evaluate them* just because the first argument is an EXPR (APPLY and APPLY* treat their argument expressions identically for both EXPR and IMPR procedures). By constructing the full 1-LISP redex, however, we are able to get to the processing decisions *before* the test is made on whether the procedure is an EXPR or IMPR.

In 2-LISP, however, having *designators* of argument expressions is just the kind of meta-structural situation in which REDUCE is appropriate:

```
(LET [[A 'X] [B 'Y]]                                                    (S4-793)
    (REDUCE '+ (RCONS A B)))     ⇒   '7         ; 2-LISP
(LET [[C '[X Y]]] (REDUCE '+ C)) ⇒   '7         ; 2-LISP
```

Although of course even in this situation REDUCE need not be used, if the intent is to remain at the object level:

```
(LET [[A 'X] [B 'Y]] (+ ↓A ↓B))  ⇒   7          ; 2-LISP      (S4-794)
(LET [[C '[X Y]]] (+ . ↓C))      ⇒   7          ; 2-LISP
```

There is a certain indisputed simplicity in the 1-LISP maxim that, when the processor evaluates a redex, it checks to see whether the function is an IMPR or an EXPR. In the former case it applies the function to the arguments without further ado; in the latter it evaluates them first, and applies the function to the values of the arguments. Other than being rather hopelessly semantical, this is not a bad characterisation of what happens. At its level of formality, furthermore, 2-LISP honours it as well — especially if one takes seriously the talk about "functions" and "applications" and "values". For consider the 2-

LISP processing of a redex. We look at the CAR of the redex and determine whether it is an intensional or extensional procedure. If intensional, we apply the function designated by that CAR *to the arguments, without further ado*; if extensional, we apply the function designated by that CAR *to the referents of the arguments.* The only thing is that, since we cannot manipulate functions explicitly, or do anything except formally simulate function application, what we *really* do is to reduce normal-form function designators with normal-form argument designators and so on and so forth.

The moral, in other words, is that 1-LISP's self-conception is not far off the mark, so long as meta-structural considerations are not taken too seriously. The problems arise — as the foregoing examples will with luck have made plain — only when one needs to make explicit reference to the structures carrying the semantical weight. It is at that point that use/mention clarity and all the rest begins to pay for the rigour it exacts.

A final set of tables will perhaps set this matter to rest once and for all. Note that we have encountered what are essentially four independent axes of decision, represented by the following four questions:

1. Do we have a standard function designator, or a *designator* of a function designator?
2. Do we have standard argument designators, or do we have *designators* of argument designators?
3. Is the function designator context relative, or global?
4. Are the arguments designated *as an objectified whole*, or piece by piece?

In 1-LISP the answer to the third question is always "global"; in 2-LISP it is always "context-relative"; this was a design choice taken long ago. Thus the dialects differ on this axis even before considering any further issues. But the remaining questions naturally form a 2×2×2 space, in which APPLY and REDUCE and so forth fill natural spots. The following tables are intended to depict the natural usage of each of the variety of simple forms in each dialect. We assume that F is a (schematic) standard function designator, that A and ARGS designates *entire sequences* of arguments, and that A_1, A_2, and so forth designate each argument individually. Similarly, FD is intended to stand in place of a term that *designates a function designator*, AD and ARGS-D are intended to *designate a sequence of argument designators*, and A_1D, A_2D, and so forth are intended to designate *individual argument designators.*

(S4-795)

		Piecewise ("spread")	*Objectified ("no-spread")*
F:	A:	(F A_1 A_2 ... A_k)	
	AD:		
FD:	A:	(APPLY* FD A_1 A_2 ... A_k)	(APPLY FD ARGS)
	AD:		(EVAL (CONS FD ARGS-D))

In contrast, the 2-LISP grid looks as follows:

(S4-796)

		Piecewise ("spread")	*Objectified ("no-spread")*
F:	A:	(F A_1 A_2 ... A_k)	(F . ARGS)
	AD:		
FD:	A:		
	AD:	(REDUCE FD [A_1D A_2D ... A_kD])	(REDUCE FD AD)

In both dialects, of course, it is possible to construct expressions that fill in the other positions. Thus we give this filled in table for 1-LISP:

(S4-797)

		Piecewise ("spread")	*Objectified ("no-spread")*
F:	A:	(F A_1 A_2 ... A_k)	(APPLY 'F ARGS)
	AD:	(EVAL (LIST 'F A_1D A_2D ... A_kD))	(EVAL (CONS 'F ARGS-D))
FD:	A:	(APPLY* FD A_1 A_2 ... A_k)	(APPLY FD ARGS)
	AD:	(EVAL (LIST FD A_1D A_2D ... A_kD))	(EVAL (CONS FD ARGS-D))

Similarly, the 2-LISP space filled in, using REDUCE explicitly:

(S4-798)

		Piecewise ("spread")	*Objectified ("no-spread")*
F:	A:	(F A_1 A_2 ... A_k)	(F . ARGS)
	AD:	(F ↓A_1 ↓A_2 ... ↓A_k)	(F . ↓ARGS)
FD:	A:	(REDUCE FD ↑[A_1 A_2 ... A_k])	(REDUCE FD ↑ARGS)
	AD:	(REDUCE FD [A_1D A_2D ... A_kD])	(REDUCE FD AD)

Equivalently, the 2-LISP space filled using down arrows rather than explicit calls to REDUCE:

(S4-799)

		Piecewise ("spread")	*Objectified ("no-spread")*
F:	A:	(F A_1 A_2 ... A_k)	(F . ARGS)
	AD:	(F ↓A_1 ↓A_2 ... ↓A_k)	(F . ↓ARGS)
FD:	A:	(↓FD A_1 A_2 ... A_k)	(↓FD . ARGS)
	AD:	(↓FD ↓A_1 ↓A_2 ... ↓A_k)	(↓FD . ↓ARGS)

There is one subtlety not brought out here: we are being careless in not distinguishing terms AD and ARG-D that *designate a series of individual argument designators*, as opposed to terms that *designate a designator of a sequence of arguments* (the difference between ['1 '2 '3] and '[1 2 3], for example). In 1-LISP these two cannot be told apart, so our confusion simply reflects its confusion. In 2-LISP these are of course distinct, but the generalisation that takes a sequence of designators to designate a sequence of entities designated, coupled with the normalisation mandate, means that the appropriate entries in these tables (the right hand column of the second and fourth rows of S4-796, S4-798, and S4-799) will in fact support both circumstances.

4.d.iii. Intensional Procedures

2-LISP has three primitive *intensional procedures* (IMPRS): IF, SET, and LAMBDA. We have explained their behaviour in the foregoing sections. It is also possible, however, to define arbitrary user intensional procedures in terms of the primitive IMPR closure, as for example in:

```
(DEFINE TEST (LAMBDA IMPR [X Y] (TYPE X)))                    (S4-800)
```

Like all inchoate reflective capabilities, we will see how IMPRS land the user in confusion regarding contexts. (As usual we will demonstrate environment difficulties, rather than control difficulties, but that is only because we haven't yet introduced any mechanisms for affecting control structure; if we had such capabilities, they would cause problems in IMPRS as well.) Nonetheless, we must explain at least to some extent how IMPRS work.

The processing (upon reduction) of the body of an ***intensional closure*** (as we will call any closure whose CAR is the primitive <IMPR> closure) is standard: the body is normalised in an environment consisting of the environment recorded in the closure (which was the environment in force when the closure was constructed) extended as dictated by the process of matching the parameter pattern against the arguments. What *distinguishes* intensional closures is that when they are reduced with arguments, the pattern is matched against a *designator of the argument expression*, rather than against the *result of normalising the argument expression.* Thus if we were to normalise the form

```
(TEST (+ 1 2) (= 1 2))                                        (S4-801)
```

then the pattern `[X Y]` would be matched against the *handle* `'[(+ 1 2) (= 1 2)]`. Because of the extended matching protocol we adopted in section 4.c.ii, this will result in the binding of X to the handle `'(+ 1 2)` and of Y to the handle `'(= 1 2)`. Thus expression S4-801 will reduce to `'PAIR`, since `(+ 1 2)` is a pair.

Before proceeding further we must arrest a potential terminological confusion. Intensional *closures* are to be distinguished from ***intensional redexes***: redexes whose CARS signify intensional closures. Additionally, an ***intensional procedure*** is a procedure whose normal-form is an intensional closure. Thus IF is an intensional procedure; therefore `(IF (= 1 2) 'YES 'NO)` is an intensional *redex* (it is not a closure at all). We similarly have *extensional closures*, *extensional procedures*, and *extensional redexes*; in section 4.d.v will

encounter the corresponding *macro closures*, *macro procedures*, and *macro redexes*.

In standard LISPS, FEXPRS and NLAMBDAS — the constructs on which 2-LISP IMPRS are based — bind parameters to their *un-evaluated* arguments. In 2-LISP we bind IMPR parameters to *designators of un-normalised* arguments, which might seem, on the face of it, to be more complex than necessary. That the argument expressions should not be *processed* is taken for granted: that is the situation intensional procedures are intended to handle. But is is not immediately clear why we need to bind to *designators* or them. It is therefore worth considering the suggestion that we simply match the IMPR pattern against the un-normalised argument expression *directly* rather than against a *designator* of it. We will reject this suggestion as incoherent, but it is instructive to see why.

Note that the acceptance of such a scheme would immediately falsify our claim that bindings are all in normal-form, since in the case at hand X would be bound directly to the redex (+ 1 2). However the fact that we have violated this aesthetic is not in itself an argument against this practice; the question would merely reduce to the utility or substance of the aesthetic claim. The question is a more serious one, about what such a circumstance would mean. Suppose the parameter X was used in the body of the intensional procedure (as indeed it is in S4-800, as an argument to TYPE). Since bindings are semantically co-referential, there can be no doubt that X would in this scheme designate the number three, but it simply isn't clear what it would mean to process X. We have said that the local procedural consequence of an atom (a parameter) is its binding; thus the local procedural consequence of X would be the redex (+ 1 2). However it would follow that processing X would not yield a normal-form designator, thereby violating the normal-form theorem, giving TYPE a structure it would not recognise, and so forth. This simply contradicts every assumption we have made about 2-LISP's Ψ.

Another possibility would be, if X was used extensionally, to have its local procedural consequence be not its binding, but the (possibly recursive) local procedural consequence of its binding. Normalising, in other words, would iterate through such bindings until a normal-form designator was achieved. Thus processing X in S4-800 would first acquire the binding of X in the local context, and then process the (+ 1 2) redex, yielding the numeral 3. This at least maintains the integrity of Ψ in one sense — in that the local procedural consequence of all terms would still be in normal form.

This plan has several consequences (and resembles in various ways ALGOL's "call-by-name" protocols). First, if an intensional parameter (a parameter in an intensional closure) were bound to an expression with side-effects, then every use of that parameter would engender the side-effects. Thus we would have, for example:

```
> ((LAMBDA IMPR [X]                          ; This is not 2-LISP,     (S4-802)
     (+ X X))                                ; but a proposed variant
   (BLOCK (PRINT 'HELLO) 4)) HELLO HELLO     ; we will soon reject.
> 8
```

This is not incoherent, but it is not minor, either. Also, there are environment problems. Suppose that "by accident", so to speak, we did this:

```
> (LET [[X 3]]                                                    (S4-803)
    ((LAMBDA IMPR [X]
       (+ X X))
     X))
```

To our possible surprise, this would cause a non-terminating computation, since X would be bound to itself, and the iterative processing scheme we are assuming would recurse forever.

Nor is this environment problem eliminable. The scheme we will have adopted for IMPRs has environment problems too, but it is easy to see from whence they stem, and it will be equally easy in 3-LISP to avoid them. Under the present scheme, however, there is no obvious way to tell what context a variable was intended to derive its significance from.

Furthermore, all of these suggestions are mechanistic in nature; they do not spring from grounded semantical argument. The essence of an intensional construct is that it derives its significance in some way from the form of the argument. What should be intensional *are the argument expressions in an intensional redex*, not the variables within the body of the intensional closure itself. *They* are standard designating variables as usual. The point, rather, is that the variables in the intensional closure should *designate* the intensional content of the argument expressions in the intensional redex. In other words, the bound parameter X in S4-802 and S4-803 should *designate* the appropriate intensional argument expression.

If in 2-LISP we had a theory whereby we could reify intensions, we might make intensional parameters designate intensions. For the time being, however, we adopted our usual hyper-intensional stance, and have them designate *expressions*. It is for this reason that we adopt the protocol we do. In 3-LISP we will bind not only the argument

expression, but the surrounding context of use; thus in 3-LISP we will be able to obtain *any level of significance* from the argument expression. Though 3-LISP will not present a theory of intension either, it will at least be able to provide coverage of the territory where such a notion might lie.

We mentioned that our IMPR scheme has context problems. To illustrate this, we will attempt (and fail) to define SET, in terms of a version of REBIND that accepted just two arguments (i.e., a two-argument REBIND will be assumed to be an extensional version of SET). We aim, that is, to define a procedure SET so that expressions of the form

```
(SET <VARIABLE> <EXPRESSION>)                                    (S4-804)
```

would be entirely equivalent to expressions of the form

```
(REBIND '<VARIABLE> ↑<EXPRESSION>)                               (S4-805)
```

Thus (SET X (+ 2 3)) should be equivalent to (REBIND 'X '5).

We begin with a plausible and certainly simple definition:

```
(DEFINE SET₁                                                     (S4-806)
   (LAMDBA IMPR [A B]
      (REBIND A (NORMALISE B))))
```

It is easy to see a problem with this definition, however: in calling NORMALISE explicitly the environment in which the expression that B designates will not be the same one that was in force when the original SET redex was normalised. In particular, two bindings — of A and B — have intervened. Thus although we might think we would correctly get:

```
(LET [[X 3]]                                  ; This actually    (S4=807)
   (BLOCK (SET₁ THREE X)                      ; won't work.
          THREE))            ⇒    3
```

it is nonetheless apparent that we would (incorrectly) generate:

```
(LET [[A 3]]                                                     (S4=808)
   (BLOCK (SET₁ THREE A)
          THREE))            ⇒    'THREE
```

and

```
(LET [[B 3]]                                                     (S4=809)
   (BLOCK (SET₁ THREE B)
          THREE))            ⇒    'B
```

The problem in S4-808 is that the binding of A to 3 is over-ridden by the subsequent

binding of A to 'THREE (the A and B of the definition of SET_1 are bound to 'THREE and 'A, respectively). Thus the interior (NORMALISE B) would return the handle on the binding of A, which is the handle ''THREE. Hence REBIND would set THREE to the handle 'THREE, unexpectedly.

In S4-809, the interior bindings of A and B would be to the handle 'THREE and 'B; thus (NORMALISE B) would return ''B; hence THREE would be bound to the handle 'B.

This example is one of the simplest ones imagineable; with just the slightest complexity in the code the unintended binding interactions in IMPRS can be virtually impossible to predict without simulating the code. Typically, the accepted practice in standard LISPS is to have definitions such as that of SET_1 use extremely unlikely spellings for their parameters, so as to minimise the chance of collision between the formal parameters of the IMPR closure and those of the expressions designated by those parameters. Thus we might expect to see a definition such as the following:

```
(DEFINE SET2                                                    (S4-810)
   (LAMDBA IMPR [##!!-SET-INTERNAL-PARAMETER-1-!!##
                 ##!!-SET-INTERNAL-PARAMETER-2-!!##]
      (REBIND ##!!-SET-INTERNAL-PARAMETER-1-!!##
              (NORMALISE ##!!-SET-INTERNAL-PARAMETER-2-!!##))))
```

As a principled solution, however, this obviously has little to recommend it. (Another standard solution — to provide IMPRS with a second argument, bound to the "calling context", is a step towards the objectification of theoretical entities that is part of reflection, to be examined in the next chapter.)

However we have an even more serious problem than this, as hinted by the comment to the side of S4-807. Completely apart from these anomolous cases, it is by no means clear how SET is *supposed* to work, given that 2-LISP is statically scoped. In 1-LISP the answer is clear, and is manifested in the way the problem identified in the previous paragraph is normally solved. Since free variables are looked up *dynamically*, we would expect the free variables in the arguments to SET to "reach back up the stack" past the bindings of A and B (or past the bindings of ##!!-SET-INTERNAL-PARAMETER-1-!!## and ##!!-SET-INTERNAL-PARAMETER-2-!!##), to their bindings in the context in which SET was called. But this betrays a hope that the call to NORMALISE in the last line of the definition of SET_2 will somehow magically use the environment in force *at the point of the call to* SET — an environment that, in a statically scoped dialect, is *no longer available once inside the body*

of the closure.

The problem, of course, stems from the switch in environments that occurs when the processor of a statically scoped language normalises the "body" of a closure. This is not a problem with a simple solution, although it does show that our first concern (with *collision* between the closure's own parameters and those in the un-normalised argument expressions) was a red-herring. One of the great benefits of statically scoped languages is that there is by and large *not* a problem of conflict across closure boundaries. Thus our imagined concern with such a collision should have alerted us to our error.

What of course we have to do is to give NORMALISE an explicit "environment" argument, obtained somehow from the underlying processor in a primitive way. Thus the last line of the definition of TEST ought rightly be (REBIND A (NORMALISE B ENV)) (we can go back to using A and B as parameters, with impunity). But there is no obvious way in which to pass such a thing to SET, unless IMPRs in general could be given the environment from the point of call *automatically.* One obvious candidate solution, namely, to provide a primitive procedure called, say, CURRENT-ENVIRONMENT, which one could call to obtain a reference to the environment currently in force, has a fatal flaw. The problem is where one would call it. If SET was called with an extra argument (i.e. (SET X (CONS A B) (CURRENT-ENVIRONMENT))), since SET is an IMPR that call wouldn't be processed, and the problem would recurse. If SET tried to execute (CURRENT-ENVIRONMENT) in *its* body, then the context of the processing of TEST's body would be returned, rather than the context of the processing of the call to TEST, which is exactly the wrong behaviour. Finally, if it were processed *outside* the scope of the call to SET, and a variable bound to the result were used within the SET redex (as for example in (LET [[ENV (CURRENT-ENVIRONMENT)]] (SET X (CONS A B) ENV))), the problem would again recurse, since there would be no way to obtain the binding of ENV.

In exploring these issues we are close to a discussion of implementing reflective procedures. It is not our mandate to suggest *how* they should be provided in this chapter; we aim merely to convince the reader from a variety of positions that *some* kind of reflective abilities are required in order to deal rigourously with standard practice. There is of course no problem in providing *primitive* intensional constructs, such as IF and LAMBDA, since we can simply *posit* that they should work in some way or other. However this

discussion of IMPRS has shown that until we have a primitive reflective capability, general intensional procedures are fraught with incurable problems.

4.d.iv. The "Up-Down" Theorem

We turn next to the proof of what we call the *up-down* theorem: a claim that both declaratively and procedurally (i.e., in terms of designation, and local and full procedural consequence) all expressions of the form ↑↓<EXP> are equivalent to (NORMALISE <EXP>). From this fact, since ↑ and ↓ are primitive functions, we can if we like excise NORMALISE from the list of 2-LISP primitives, since we have a way of defining it. The theorem has a corrollary with respect to REDUCE; we said in section 4.d.ii that REDUCE could be defined in terms of NORMALISE, but it is also true that we can reduce it to a combination of up and down arrows as well. In particular, any expression of the form (REDUCE <E_1> <E_2>) will be entirely equivalent to one of the form ↑(↓<E_1> . ↓<E_2>). Put informally, these two results can be stated as follows:

(NORMALISE S) ≡ ↑↓S (S4-814)

(REDUCE S_1 S_2) ≡ ↑(↓S_1 . ↓S_2) (S4-815)

More formally, however, we have the following characterisation of the first of these (this is the mathematical statement we will prove):

$$\forall S_1,S_2,S_3 \in \textit{ATOMS},\ S_4 \in \textit{S},\ E \in \textit{ENVS},\ F \in \textit{FIELDS},\ C \in \textit{CONTS} \qquad \text{(S4-816)}$$
$$[[[\,E(S_1) = E_0(\text{"}\textit{NORMALISE})\,] \wedge$$
$$[\,E(S_2) = E_0(\text{"}\textit{NAME})\,] \wedge$$
$$[\,E(S_3) = E_0(\text{"}\textit{REFERENT})\,]] \supset$$
$$[\,\Sigma(\text{"}(\underline{S_1}\ \underline{S_4}),E,F,C) = \Sigma(\text{"}(\underline{S_2}\ (\underline{S_3}\ \underline{S_4})),E,F,C)\,]]$$

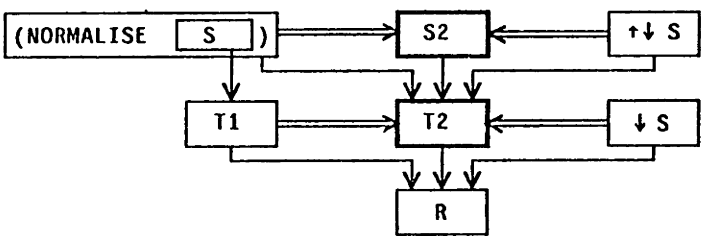
Similarly, the corollary has a similar formal statement:

$$\forall S_1,S_2,S_3 \in \textit{ATOMS},\ S_4,S_5 \in \textit{S},\ E \in \textit{ENVS},\ F \in \textit{FIELDS},\ C \in \textit{CONTS} \qquad \text{(S4-817)}$$
$$[[[\,E(S_1) = E_0(\text{"}\textit{REDUCE})\,] \wedge$$
$$[\,E(S_2) = E_0(\text{"}\textit{NAME})\,] \wedge$$
$$[\,E(S_3) = E_0(\text{"}\textit{REFERENT})\,]] \supset$$
$$[\,\Sigma(\text{"}(\underline{S_1}\ \underline{S_4}\ \underline{S_5}),E,F,C) = \Sigma(\text{"}(\underline{S_2}\ ((\underline{S_3}\ \underline{S_4})\ .\ (\underline{S_3}\ \underline{S_5})),E,F,C)\,]]$$

Before we set out to prove this, it is important to realise that this is a different result from the less formal conclusion argued throughout this chapter, and summarised in section 4.h: that there is very little need ever to *use* NORMALISE explicitly (be it primitive or derived): that many of the traditional reasons one needs access to such a function are handled directly in the 2-LISP base language, without any need of meta-structural facilities at all.

The diagram given in S4-818 below shows why the result is true. In particular, for any expression S, the term (NORMALISE S) *designates* what (REFERENT S) *normalises to.* The point is that the referent of (NORMALISE S) is $\Psi\Phi(S)$ — Φ because NORMALISE is an extensional function; Ψ because NORMALISE designates EXT(Ψ). This was also the essential content of diagram S4-768. On the other hand, the normal-form of (REFERENT S) is $\Psi\Phi(S)$, indicated below but also depicted in S4-735. Therefore the normal-form of (NORMALISE S) is $\Phi^{-1}\Psi\Phi(S)$, where Φ^{-1} is the HANDLE function since the range of Ψ is *S*. Similarly, the normal-form of (NAME S) (the expansion of ↑S) is $\Phi^{-1}\Psi(S)$. Hence the normal form of (NAME (REFERENT S)) is $\Phi^{-1}\Psi\Psi\Phi(S)$, which collapses to $\Phi^{-1}\Psi\Phi(S)$, since Ψ is idempotent. Thus the two are equivalent.

(S4-818)



We will prove only S4-816; the proof of S4-817 is entirely parallel. The technique will be to expand the significance of each side of the equation, using the preconditions as premises (i.e. using the deduction theorem). We start with the "(NORMALISE S)" side. Assuming that

$$[[S_1,S_2,S_3 \in \textit{ATOMS}] \wedge [S_4 \in \textit{S}] \wedge [E \in \textit{ENVS}] \wedge [F \in \textit{FIELDS}] \wedge [C \in \textit{CONTS}] \wedge [E(S_1) = E_0(\text{"}\textit{NORMALISE}\text{"})] \wedge [E(S_2) = E_0(\text{"}\textit{NAME}\text{"})] \wedge [E(S_3) = E_0(\text{"}\textit{REFERENT}\text{"})]] \quad \text{(S4-819)}$$

we look at

$$\Sigma(\text{"}(\underline{S_1}\ \underline{S_4}),E,F,C) \quad \text{(S4-820)}$$

Because of the significance of pairs (S4-38) this reduces to:

$$= \Sigma(S_1,E,F,[\lambda\langle S_2,D_2,E_2,F_2\rangle . [\Delta S_2(\text{"}[\underline{S_4}],E_2,F_2,[\lambda\langle S_3,E_3,F_3\rangle . C(S_3,[D_2(\text{"}[\underline{S_4}],E_2,F_2)],E_3,F_3)])]]) \quad \text{(S4-821)}$$

But we know the full significance of S_1 (from S4-819); hence we get (we have performed a variety of α-reductions, even those not strictly mandated, to make this slightly clearer):

```
= ([λE5,λF5,λC5 . C5("(<EXPR> E0 '[TERM] '(NORMALISE TERM)),                (S4-822)
                    [λS6,E6,F6 . Σ(S6,E6,F6,
                                   [λ<S7,D7,E7,F7> .
                                       Σ(NTH(1,D7,F7),E7,F7,
                                         [λ<S8,D8,E8,F8> . S8])])]
                    E5,F5)]
    <E,
     F,
     [λ<S2,D2,E2,F2> .
         [ΔS2("[S4],E2,F2,
              [λ<S3,E3,F3> . C(S3,[D2("[S4],E2,F2)],E3,F3)])]]>
```

Reducing this once:

```
=([λ<S2,D2,E2,F2> .                                                         (S4-823)
         [ΔS2("[S4],E2,F2,
              [λ<S3,E3,F3> . C(S3,[D2("[S4],E2,F2)],E3,F3)])]]
   <"(<EXPR> E0 '[TERM] '(NORMALISE TERM)),
    [λS6,E6,F6 . Σ(S6,E6,F6,
                  [λ<S7,D7,E7,F7> .
                      Σ(NTH(1,D7,F7),E7,F7,[λ<S8,D8,E8,F8> . S8])])]
    E,F>)
```

and again:

```
= ([Δ"(<EXPR> E0 '[TERM] '(NORMALISE TERM))]                                (S4-824)
   <"[S4],E,F,[λ<S3,E3,F3> .
              C(S3,([λS6,E6,F6 .
                      Σ(S6,E6,F6,
                          [λ<S7,D7,E7,F7> .
                              Σ(NTH(1,D7,F7),E7,F7,
                                [λ<S8,D8,E8,F8> . S8])])]
                    <"[S4],E,F>),
                   E3,
                   F3)]>)
```

Before applying the internalised NORMLALISE, it is convenient to simplify the continuation:

```
= ([Δ"(<EXPR> E0 '[TERM] '(NORMALISE TERM))]                                (S4-825)
   <"[S4],E,F,[λ<S3,E3,F3> .
              C(S3,[Σ("[S4],E,F,
                      [λ<S7,D7,E7,F7> .
                          Σ(NTH(1,D7,F7),E7,F7,[λ<S8,D8,E8,F8> . S8])])],
                   E3,
                   F3)]>)
```

Now we know the internalisation of the primitive NORMALISE closure from S4-776; hence we can expand this into:

```
= ([λ<S2,E2,F2,C2> .                                                    (S4-826)
     Σ(S2,E2,F2,
       [λ<S3,D3,E3,F3> .
         Σ(HANDLE⁻¹(NTH(1,S3,F3)),E3,F3,
           [λ<S9,D9,E9,F9> . C2(HANDLE(S9),E9,F9)])])]
   <"[S4],E,F,[λ<S3,E3,F3> .
               C(S3,[Σ("[S4],E,F,
                      [λ<S7,D7,E7,F7> .
                        Σ(NTH(1,D7,F7),E7,F7,[λ<S6,D8,E8,F8> . S8])])],
                 E3,
                 F3)]>)
```

And reduce:

```
= Σ("[S4],E,F,                                                          (S4-827)
    [λ<S3,D3,E3,F3> .
      Σ(HANDLE⁻¹(NTH(1,S3,F3)),E3,F3,
        [λ<S9,D9,E9,F9> .
          ([λ<S3,E3,F3> .
             C(S3,[Σ("[S4],E,F,
                    [λ<S7,D7,E7,F7> .
                      Σ(NTH(1,D7,F7),E7,F7,
                        [λ<S8,D8,E8,F8> . S8])])],
               E3,
               F3)]
           <HANDLE(S9),E9,F9>)])])
```

Now rather than demonstrate all the intervening steps involved in establishing the significance of the rail "[S4], we can convert this to a simple question of the significance of S_4 on its own:

```
= Σ(S4,E,F,                                                             (S4-828)
    [λ<S3,D3,E3,F3> .
      Σ(HANDLE⁻¹(S3),E3,F3,
        [λ<S9,D9,E9,F9> .
          ([λ<S3,E3,F3> .
             C(S3,[Σ(S4,E,F,
                    [λ<S7,D7,E7,F7> .
                      Σ(D7,E7,F7,
                        [λ<S8,D8,E8,F8> . S8])])],
               E3,
               F3)]
           <HANDLE(S9),E9,F9>)])])
```

We can collapse the continuation:

```
= Σ(S4,E,F,                                                             (S4-829)
    [λ<S3,D3,E3,F3> .
      Σ(HANDLE⁻¹(S3),E3,F3,
        [λ<S9,D9,E9,F9> .
          C(HANDLE(S9),
            [Σ(S4,E,F,[λ<S7,D7,E7,F7> .
                       Σ(D7,E7,F7,[λ<S8,D8,E8,F8> . S8])])],
            E9,
```

$F_9)]))]$

This is approximately what we would expect: the structure S_4 would first be processed, yielding a handle S_3. The *referent* of this handle ($\text{HANDLE}^{-1}(S_3)$) would then in turn be processed, after which the handle designating what *it* returned would be given to the original caller. However note that this too can be drastically simplified. If S_3 is a handle, as the equation demands it must be, then D_3 must *equal* $\text{HANDLE}^{-1}(S_3)$. Hence the embedded designational function is equivalent to the overall function in which it is embedded (i.e. $D_7 = D_3 = \text{HANDLE}^{-1}(S_3)$); hence S4-829 can be collapsed down to:

$$= \Sigma(S_4,E,F, \quad \text{(S4-830)}$$
$$[\lambda\langle S_3,D_3,E_3,F_3\rangle .$$
$$\Sigma(\text{HANDLE}^{-1}(S_3),E_3,F_3,$$
$$[\lambda\langle S_9,D_9,E_9,F_9\rangle . C(\text{HANDLE}(S_9),S_9,E_9,F_9)])])$$

A clearer account is hard to imagine.

This is half of the proof; the other proceeds similarly; we will therefore present only some of the intervening steps. We start with the same assumptions, and look for the appropriate expansion of:

$$\Sigma(\text{"}(\underline{S_2}\ (\underline{S_3}\ \underline{S_4})),E,F,C) \quad \text{(S4-831)}$$

Again, being a pair, this reduces to:

$$= \Sigma(S_2,E,F, \quad \text{(S4-832)}$$
$$[\lambda\langle S_1,D_1,E_1,F_1\rangle .$$
$$[\Delta S_1(\text{"}[(\underline{S_3}\ \underline{S_4})],E_1,F_1,$$
$$[\lambda\langle S_3,E_3,F_3\rangle . C(S_3,[D_1(\text{"}(\underline{S_3}\ \underline{S_4}),E_1,F_1)],E_3,F_3)])]])$$

Taking the significance of S_2 from S4-819 (since we know that S_2 bound to the primitive NAME closure):

$$= ([\lambda E_5,\lambda F_5,\lambda C_5 . C_5(\text{"}\langle EXPR\rangle\ E_0\ '[TERM]\ '(NAME\ TERM)), \quad \text{(S4-833)}$$
$$[\lambda S_6,E_6,F_6 .$$
$$\Sigma(S_6,E_6,F_6,[\lambda\langle S_7,D_7,E_7,F_7\rangle . NTH(1,S_7,F_7)])]$$
$$E_5,F_5)]$$
$$\langle E,$$
$$F,$$
$$[\lambda\langle S_1,D_1,E_1,F_1\rangle .$$
$$[\Delta S_1(\text{"}[(\underline{S_3}\ \underline{S_4})],E_1,F_1,$$
$$[\lambda\langle S_3,E_3,F_3\rangle . C(S_3,[D_1(\text{"}[(\underline{S_3}\ \underline{S_4})],E_1,F_1)],E_3,F_3)])]]\rangle$$

A few simple reductions:

$$= ([\lambda\langle S_1,D_1,E_1,F_1\rangle . \quad \text{(S4-834)}$$
$$[\Delta S_1(\text{"}[(\underline{S_3}\ \underline{S_4})],E_1,F_1,$$
$$[\lambda\langle S_3,E_3,F_3\rangle . C(S_3,[D_1(\text{"}[(\underline{S_3}\ \underline{S_4})],E_1,F_1)],E_3,F_3)])]]$$

```
    <("(<EXPR> E0 '[TERM] '(NAME TERM)),
     [λS6,E6,F6 . Σ(S6,E6,F6,[λ<S7,D7,E7,F7> . NTH(1,S7,F7)])],
     E,F>)

  = ([Δ("(<EXPR> E0 '[TERM] '(NAME TERM)))]                          (S4-835)
    <"[(S3 S4)],
     E,
     F,
     [λ<S3,E3,F3> .
         C(S3,
           ([λS6,E6,F6 . Σ(S6,E6,F6,[λ<S7,D7,E7,F7> . NTH(1,S7,F7)])]
            <"[(S3 S4)],E,F>),
           E3,F3)]>)

  = ([Δ("(<EXPR> E0 '[TERM] '(NAME TERM)))]                          (S4-836)
    <"[(S3 S4)],
     E,
     F,
     [λ<S3,E3,F3> .
         C(S3,
           Σ("[(S3 S4)],E,F,[λ<S7,D7,E7,F7> . NTH(1,S7,F7)]),
           E3,F3)]>)
```

Now we obtain the internalised NAME function from S4-769:

```
  = ([λ<S1,E1,F1,C1> .                                               (S4-837)
         Σ(S1,E1,F1,[λ<S2,D2,E2,F2> . C1(HANDLE(NTH(1,S2,F2)),E2,F2)])]
    <"[(S3 S4)],
     E,
     F,
     [λ<S3,E3,F3> .
         C(S3,
           Σ("[(S3 S4)],E,F,[λ<S7,D7,E7,F7> . NTH(1,S7,F7)]),
           E3,F3)]>)
```

As before, we will intervene in this to simplify the processing of sequences — we convert it to a single argument format:

```
  = ([λ<S1,E1,F1,C1> .                                               (S4-838)
         Σ(S1,E1,F1,[λ<S2,D2,E2,F2> . C1(HANDLE(NTH(1,S2,F2)),E2,F2)])]
    <"(S3 S4),
     E,
     F,
     [λ<S3,E3,F3> .
         C(S3,
           Σ("(S3 S4),E,F,[λ<S7,D7,E7,F7> . NTH(1,S7,F7)]),
           E3,F3)]>)
```

Applying the internalised function:

```
  = Σ("(S3 S4),                                                      (S4-839)
      E,
      F,
      [λ<S2,D2,E2,F2> .
          ([λ<S3,E3,F3> .
```

```
        C(S3,Σ("(S3 S4),E,F,[λ<S7,D7,E7,F7> . NTH(1,S7,F7)]),E3,F3)]
    <HANDLE(NTH(1,S2,F2)),E2,F2>)])
```

Note that we have now discharged the "NAME" portion of (NAME (REFERENT X)); what remains is the signficance of (REFERENT X), with a revised continuation that, as expected, from a procedural point of view constructs a handle, and from a declarative point of view designates what (REFERENT X) will return. For the second last time we expand this in terms of the general significance of pairs:

```
= Σ(S3,E,F,                                                          (S4-840)
    [λ<S1,D1,E1,F1>
        [Δ(S1)]("[S4],E1,F1,
                [λ<S2,E2,F2> .
                    ([λ<S5,D5,E5,F5> .
                        ([λ<S6,E6,F6> .
                            C(S6,
                              Σ("(S3 S4),E,F,
                                [λ<S7,D7,E7,F7> . NTH(1,S7,F7)]),
                              E6,
                              F6)]
                         <HANDLE(NTH(1,S5,F5)),E5,F5>)]
                 <S2,[D1("[S4],E1,F1)],E2,F2>)
```

There are two internal reductions that can be discharged:

```
= Σ(S3,E,F,                                                          (S4-841)
    [λ<S1,D1,E1,F1>
        [Δ(S1)]("[S4],E1,F1,
                [λ<S2,E2,F2> .
                    C(HANDLE(NTH(1,S2,F2)),
                      Σ("(S3 S4),E,F,[λ<S7,D7,E7,F7> . NTH(1,S7,F7)]),
                      E2,
                      F2)])])
```

We threw away one designated entity, but we have to re-expand the significance of (S3 S4) one more time:

```
= Σ(S3,E,F,                                                          (S4-842)
    [λ<S1,D1,E1,F1>
       [Δ(S1)]("[S4],E1,F1,
               [λ<S2,E2,F2> .
                   C(HANDLE(NTH(1,S2,F2)),
                     Σ(S3,E,F,
                       [λ<S1,D1,E1,F1>
                           [Δ(S1)]("[S4],E1,F1,
                                   [λ<S2,E2,F2> .
                                      ([λ<S7,D7,E7,F7> . NTH(1,S7,F7)]
                                       <S2,[D1("[S4],E1,F1)],E2,F2>)])]),
                     E2,
                     F2)])])
```

However as usual there is a great simplication that can be treated here. There are two identical structures obtaining the significance of $(S_3\ S_4)$; they can be collapsed:

```
= Σ(S3,E,F,                                                          (S4-843)
    [λ<S1,D1,E1,F1>
       [Δ(S1)]("[S4],E1,F1,
               [λ<S2,E2,F2> .
                   C(HANDLE(NTH(1,S2,F2)),
                     [Δ(S1)]("[S4],E1,F1,
                             [λ<S2,E2,F2> .
                                 ([λ<S7,D7,E7,F7> . NTH(1,S7,F7)]
                                  <S2,[D1("[S4],E1,F1)],E2,F2>)]),
                     E2,
                     F2)])])
```

We can also perform a reduction in the internal continuation:

```
= Σ(S3,E,F,                                                          (S4-844)
    [λ<S1,D1,E1,F1>
       [Δ(S1)]("[S4],E1,F1,
               [λ<S2,E2,F2> .
                   C(HANDLE(NTH(1,S2,F2)),
                     [Δ(S1)]("[S4],E1,F1,[λ<S2,E2,F2> . NTH(1,S2,F2)]),
                     E2,
                     F2)])])
```

And again dispense with the redundancy of using the internalised REFERENT function twice:

```
= Σ(S3,E,F,                                                          (S4-845)
    [λ<S1,D1,E1,F1>
       [Δ(S1)]("[S4],E1,F1,
               [λ<S2,E2,F2> .
                   C(HANDLE(NTH(1,S2,F2)),NTH(1,S2,F2),E2,F2)])])
```

We are next ready to obtain the full significance of the primitive REFERENT closure from S4-763:

```
= ([λ<S1,D1,E1,F1>                                                   (S4-846)
      [Δ(S1)]("[S4],E1,F1,
              [λ<S2,E2,F2> .
                  C(HANDLE(NTH(1,S2,F2)),NTH(1,S2,F2),E2,F2)])]
   <"(<EXPR> E0 '[TERM] '(REFERENT TERM)),
    [λ<S1,E1,F1> .
        Σ(S1,E1,F1,
          [λ<S2,D2,E2,F2> .
              Σ(NTH(1,D2,F2),E2,F2,[λ<S3,D3,E3,F3> . D3])])],
    E,F>)
```

We begin our final set of substitutions:

```
= ([Δ("(<EXPR> E0 '[TERM] '(REFERENT TERM)))]                        (S4-847)
   <"[S4],E,F,[λ<S2,E2,F2> .
                  C(HANDLE(NTH(1,S2,F2)),NTH(1,S2,F2),E2,F2)]>)
```

Note that the *designation* of (REFERENT X) has just been thrown away, which is quite proper: (NAME (REFERENT X)) is going to designate what (REFERENT X) *returns*, and will return a handle designating that result; hence the referent of (REFERENT X) is immaterial in this circumstance. We pick up the internalised REFERENT function, leading to this:

```
= ([λ<S1,E1,F1,C1> .                                                    (S4-848)
     Σ(S1,E1,F1,
       [λ<S6,D6,E6,F6> .
            Σ(HANDLE⁻¹(NTH(1,S6,F6)),E6,F6,
              [λ<S3,D3,E3,F3> . C1(S3,E3,F3)])])]
   <"[S4],E,F,[λ<S2,E2,F2> .
                C(HANDLE(NTH(1,S2,F2)),NTH(1,S2,F2),E2,F2)]>)
```

Substituting:

```
= Σ("[S4],E,F,                                                          (S4-849)
    [λ<S6,D6,E6,F6> .
        Σ(HANDLE⁻¹(NTH(1,S6,F6)),
          E6,
          F6,
          [λ<S3,D3,E3,F3> .
              ([λ<S2,E2,F2> .C(HANDLE(NTH(1,S2,F2)),NTH(1,S2,F2),E2,F2)]
               <S3,E3,F3>)])])
```

Our standard technique of converting to a single argument:

```
= Σ(S4,E,F,                                                             (S4-850)
    [λ<S6,D6,E6,F6> .
        Σ(HANDLE⁻¹(S6),
          E6,
          F6,
          [λ<S3,D3,E3,F3> .
              ([λ<S2,E2,F2> .C(HANDLE(S2),S2,E2,F2)]
               <S3,E3,F3>)])])
```

And reducing:

```
= Σ(S4,E,F,                                                             (S4-851)
    [λ<S6,D6,E6,F6> .
        Σ(HANDLE⁻¹(S6),E6,F6,
          [λ<S3,D3,E3,F3> .C(HANDLE(S3),S3,E3,F3)])])
```

But this is exactly the same as S4-830. Hence the two sides of the equation in S4-817 have been shown identical. One use of the deduction theorem, then, gives us S4-817. *Q.E.D.*

Given this result, and the attendant corrollary, we could now *define* NORMALISE and REDUCE as follows:

```
(DEFINE NORMALISE (LAMBDA EXPR [S] ↑↓S))                                (S4-852)
```

```
(DEFINE REDUCE (LAMBDA EXPR [S1 S2] ↑(↓S1 . ↓S2)))          (S4-863)
```

However we will not do this, since we are about to shift to 3-LISP, where the results will no longer be true, since NORMALISE and REDUCE will be given expanded roles to play. What is interesting about these results, however, is that we can now use the up and down arrows to effect any behaviour that would *in 2-LISP* have been obtained using NORMALISE and REDUCE. The simple cases will remain simple, in other words, which is a pleasant result.

Furthermore, it was instructive to have defined NORMALISE and REDUCE on their own initially, since it is only with an *independent* definition of their significance that we have been able to show, with any confidence or insight, that appropriate combinations of "↑" and "↓" adequately discharge their particular responsibilities.

4.d.v. Macros and Backquote

The discussion in section 4.d.iii made it clear that at least in this dialect non-primitive intensional procedures of the IMPR form were of dubious value, since they had lost any connection with the context under which the intensional redex was originally processed. In this section we look at ***macros*** — a different kind of intensional procedure that partially circumvents this particular difficulty. Though macros will be definable in 3-LISP as a certain special type of reflective procedure, in 2-LISP they must be primitive for much the same reason as IMPRS are limited: they involve a re-use of the environment in force at the point of their original reduction.

Macros are, informally speaking, procedures that designate functions from structure to structure. The idea is that when a ***macro redex*** (a redex whose CAR signifies a macro procedure) is reduced, the macro procedure signified by the CAR of the redex constructs a different structure out of the "argument expressions", to be processed in place of the original redex. For example, we can easily define a macro procedure INCREMENT so that any redex of the form (INCREMENT <EXPRESSION>) will be converted into one of the form (+ 1 <EXPRESSION).

Of course the INCREMENT in (INCREMENT (* X Y)) can't quite be said to designate a function that transforms the rail [(* X Y)] to the redex (+ 1 (* X Y)), since processing (INCREMENT (* X Y)) will not only construct this further redex, but will then process it as well. The processing of macros, in other words, naturally falls into two rather distinct parts: a ***first phase*** computation that yields what is often called the "expanded" form, and then a ***second phase*** that processes that expanded form in the standard fashion. Declaratively, then, INCREMENT will have to be shown to designate the simple successor function; *procedurally*, however, it will involve these two computational parts, the first of which is a structure-transforming operation. As we will ultimately see in detail, these two parts are best seen as happening at distinct semantic levels.

There are a variety of subtleties arising in connection with macros, having to do in part with the following issues:

1. The interaction just mentioned between the context in which the "translation" is effected, and the context in which the resultant expression is then processed;

2. The interaction between non-rail CDR's (objectifying mutliple-arguments in the source redex) and the patterns of macro closures;
3. The use of recursion in macro definitions; and
4. The allowable dependencies of the translation process on the context specific significance of the form being translated.

Before taking up these issues, however, we do well to illustrate some simple cases. Like IMPRS, macros are normally constructed in 2-LISP using the atom MACRO (which is bound in the initial environment to the primitive <MACRO> closure, similar in structure to the <EXPR> and <IMPR> closures) in the "type" argument position in a LAMBDA expression. Thus in order to define the INCREMENT macro just mentioned, we might use something like the following (using the redex-constructing XCONS defined in S4-313):

```
(DEFINE INCREMENT1                                              (S4-860)
   (LAMBDA MACRO [X] (XCONS '+ '1 X)))
```

This definition works because, as with IMPRS, the formal parameters in a MACRO procedure are bound to *designators* of the argument expressions in the macro redex. If for example we normalised

```
(LET [[A 5] [B 6]]                                              (S4-861)
    (INCREMENT1 (* A B)))
```

the parameter X in the pattern of INCREMENT_1 would be bound to the handle '(* A B), thus designating the * redex. Consequently, the body expression (XCONS '+ '1 X) would designate (+ 1 (* A B)). In general the structure that is *designated* by the body of the closure signified by the macro redex — designated, in our example, by (XCONS '+ '1 X), in other words — is then processed *in approximately the same context as the original macro redex.* In our example, for instance, (+ 1 (* A B)) would be processed in the context where A was bound to 5 and B to 6. Thus S4-861 would normalise to 31, as expected.

We say "approximately" for two reasons. First, as always, the *field* component of the context is passed through serially from one normalisation to the next; thus if normalising the body of the macro procedure affects the field, those changes will be visible to the subsequent processing of the expression returned by the macro. The following expression, in other words, would designate the atom C, not the atom A:

```
(LET [[X '(A . B)]]                                             (S4-862)
    (LABEL [[TEST (LAMBDA MACRO [Y]
                    (BLOCK (RPLACA X 'C) (XCONS 'CAR Y)))]]
        (TEST X)))
```

In addition, to the extent that the macro affects environment structures that it shares with other procedures, it may alter subsequent processing of those procedures (*that* it shares environment follows from standard static scoping protocols, and is evidenced by the fact that the TEST macro in the preceding example retained the binding of X). This means that the context in which the expression returned by the macro redex is normalised may not be identical to that in which the macro redex itself was processed. Thus the following (a variant on S4-862) would also designate C rather than A:

```
(LET [[X '(A . B)]]                                                    (S4-863)
    (LABEL [[TEST (LAMBDA MACRO [Y]
                    (BLOCK (SET X '(C . D)) (XCONS 'CAR Y)))]]
          (TEST X)))
```

Both of these behaviours, however, are non-standard in the sense that they are rarely utilised. Much more common are the simple kinds of macro expansions exemplified by the definition of INCREMENT$_1$ above.

It is evident, in this description, that in defining a macro what one provides is the "code" for only the *first phase* of the processing of macros; the second phase — the processing *of* the structure designated by the first phase, follows normal rules. In fact it is easiest to think of macros in the following fashion: upon encountering a macro redex, the normal processing is interrupted, and a computation of a rather different sort is enjoined, which runs around and constructs an appropriate expression, based presumably on the form of arguments in the macro redex, and perhaps on other things as well. When this expression has been constructed, it is handed back to the processor, as if with the comment "*OK, I've got the expression you <u>really</u> want to process; you can resume now*".

When viewed in this manner, macros look to be procedures that, like the processor itself, sit one level above the structures under interpretation, manipulating them in various ways (but always *formally*, of course). Whereas the regular processing algorithms are general and uniform in application, redexes that invoke macros provide a way in which *special purpose* programs can run. This of course is inchoate reflection: our general characterisation of reflective procedures will be of code that runs at the same level as the regular processor, integrated with that processor in ways that the next chapter will make clear. What distinguishes macros from more general reflective procedures is this simple fact: whereas a reflective procedure can in the general case engender any computation — can engage, roughly speaking, in any dialogue whatsoever with the normal processor — a

macro procedure engages in a *particular and constrained form of dialogue*: namely, one that ends with the macro saying the sentence ending the previous paragraph, to the effect that the processor may continue, in essentially the same state that it was *before* encountering the macro redex, with a new form to process.

Strikingly, the structure of this conversation will be manifested rather clearly in the definition of MACRO in chapter 5. In our present dialect, however, MACROS have to be primitive, because we have no sufficiently powerful protocol in which to define such a constrained interaction. There are, however, a variety of properties of (and difficulties with) macros that can be illustrated here. Before turning to them, however, we need to pause for a digression, and introduce the 2-LISP backquote notation, for a very simple reason: without it the definition of any but the most trivial macros becomes almost unmanageable. We will therefore put the discussion of macros themselves aside for a few pages.

The "back-quote" notational extension we will adopt is not unlike that of 1-LISP, modified to fit 2-LISP's notational and semantical conventions. In 1-LISP, we said that expressions of the form `<EXP> were equivalent in procedural consequence to those of the form '<EXP>, except that occurences within <EXP> of forms preceded by a comma would be *evaluated* when the whole expression was evaluated. Thus we had, where X had the value 3 and Y the value NIL (this is 1-LISP):

```
`(+ 4 ,X)                          →   (+ 4 3)                        (S4-864)
`(CONS ',X ',(CONS 'A Y))          →   (CONS '3 '(A))
`(CONS ',X ,'(CONS 'A Y))          →   (CONS '3 (CONS 'A Y))
(EVAL `(+ 4 ,X))                   →   7
(EVAL `(CONS ',X ',(CONS 'A Y)))   →   (3 A)
```

1-LISP's backquote, in other words, was defined in terms of *evaluation*, whereas we will have to define expressions containing back-quotes in terms both of designation and procedural consequence. Since evaluation is a notion we have pulled apart into two notions of *normalisation* and *de-referencing*, we have to decide whether a comma in a 2-LISP's back-quoted expression should imply that the expression it precedes should be *normalised* or *de-referenced* when inserted into the whole. The two different candidates have observably different consequences. The following would be implied, in particular, if we take it to imply normalisation (assume that X is bound to 3, Y to [$T $F], and W to '[1 2]):

```
`(+ 4 ,X)                          ⇒  '(+ 4 3)                    (S4=866)
`(AND . ,Y)                        ⇒  '(AND $T $F)
`(+ . ,W)                          ⇒  '(+ . '[1 2])
(NORMALISE `(+ 4 ,X))              ⇒  '7
(NORMALISE `(AND . ,Y))            ⇒  '$F
(NORMALISE `(+ . ,W))              ⇒  <ERROR: Pattern failure>
(NORMALISE `(+ . ,↓W))             ⇒  '3
(NTH 2 ↓`(PREP 'O ,W))             ⇒  1
```

If we take the comma to imply de-referencing, on the other hand, we would have (assuming the same bindings):

```
`(+ 4 ,X)                          ⇒  <ERROR: expected an s-expr> (S4=866)
`(AND . ,Y)                        ⇒  <ERROR: expected an s-expr>
`(+ . ,W)                          ⇒  '(+ 1 2)
`(+ 4 ,↑X)                         ⇒  '(+ 4 3)
(NORMALISE `(+ . ,W))              ⇒  '3
(NTH 2 ↓`(PREP 'O ,W))             ⇒  <ERROR: expected an s-expr>
(NTH 2 ↓`(PREP 'O ,↑W))            ⇒  1
```

In both S4-865 and S4-866 the variables X and Y are bound to designators of mathematical objects (a numeral and a sequence, respectively), whereas the variable W is bound to a designator of a structural rail. In S4-865, where the comma implies that the *normal-form* is to be used, the first two examples yield valid structures; the third yields a legal structure, but one that causes a semantic error upon normalisation (as the sixth line demonstrates). In S4-866, on the other hand, the first and second examples yield processing errors, since a number cannot be part of a pair; the third, however, under this regime yields a semantically well-formed addition redex. The fourth line illustrates a repair to the example of the first line by using the explicit naming operator (↑).

It should be clear that both alternatives are well-defined, and both usable: as the examples show, an explicit naming operator can be used to overcome the automatic de-referencing in the second scheme, and an entirely parallel strategy can be used under the first scheme to de-reference explicitly when that is required. The question in deciding between them reduces to a question of whether in our use of such notation we think of the expressions preceded by commas as *designating the expression that should form a constituent in the whole*, or whether we think of it is a kind of *variable* or *schematic* constituent, one that *designates what the constituent in the whole should designate, relativised to circumstance* (the former is the de-referencing alternative; the latter the normalising one). Although

back-quotes in general form expressions, which might seem to argue for the former choice, it is a fact about quoted expressions that once one has writen the quote, one then writes the symbols that form its constituents *as if one were using* (not mentioning) *them.* For example, in writing the lexical notation that notates the structure that designates the pair consisting of the symbols "+", "2", and "3", we write `'(+ 2 3)`, not `'('+ '2 '3)`. A single quote, in other words, suffices for the entire expression within its scope. The question, then, is whether we think of an expression preceded by a comma as being *within* the scope of the quotation, or without it. The basic power of the back-quote notation is that it enables us to think *as if we were using structure*, not about how to designate it, even though the structure that the back-quote notation actually notates is structure designating. It would seem to follow, therefore, that the comma should not itself de-reference, since we will have performed the requisite degree of de-referencing ourselves. However it is also true that the structure within the closer scope of the comma has to do with specifying what expression should form the contituent; it is not structure of that constituent.

It is not clear that a unique and principled answer is forthcoming. We will adopt the first alternative: that expressions within the scope of a comma should *designate* the consituent to be used in the overall quoted expression. We will view these "comma'ed" expressions, in other words, as *structure designators*, not as *schematic terms.* This facilitates macro definitions, which is our present subject matter: we would thus define the (simple version) of the INCREMENT$_1$ macro as follows:

```
(DEFINE INCREMENT1                                          (S4-867)
   (LAMBDA MACRO [X]
      `(+ ,X 1)))
```

Under the scheme we are rejecting this would have to be defined as follows:

```
(DEFINE INCREMENT1                                          (S4-868)
   (LAMBDA MACRO [X]
      `(+ ,↓X 1)))
```

which seems less compelling. This does not, however, seem so much a principled as a pragmatic choice.

To express this decision precisely requires a little care, since we have to speak expclitly both of notation and of designation. We can summarise it as follows, in what we will call the ***back-quote principle***:

> *A lexical expression E_1 preceded by a back-quote will notate a structure S_1 that designates a structure S_2 that would be notated by E_1, with the exception that those fragments of S_2 that would be notated by portions of E_1 that are preceded by a comma will in fact be designated by the structure that those portions notate, rather than notated by them directly.*

This can be understood using our example. In S4-867, the expression E_1 is "(+ ,X 1)"; since it is preceded by a back-quote, it will notate a structure S_1 that designates a structure S_2 with certain properties. In particular, S_2 would be notated by "(+ ,X 1)" — would, in other words, be a "+" redex, except that the one portion of S_2 that would be notated by the portion of E_1 that is preceded by a comma — the first element of the rail that is S_2's CDR, in other words — is not in fact notated by "X", but is instead *designated* by the structure notated by "X". We know this, in other words: S_2 will be a redex whose CAR is the atom + and whose CDR is a two-element rail. The second element of that rail will be the numeral 1, but we don't know *exactly* what the first element will be; all we know is that it will be *designated* by the atom X.

An obvious S_1 satisfying this account is this:

```
(PCONS '+ (RCONS X '1))                                    (S4-869)
```

Thus S4-869 is a candidate for what "`(+ ,X 1)" notates. It should be noted, however, that the back-quote principle is not completely specific as to what structure a given back-quoted expression will notate: the constraint is entirely on its designation. Thus the following would also be allowed (given as suitable definition of SUBST):

```
(SUBST X '??? '(+ ??? 1))                                  (S4-870)
```

although this of course suggests an unworkable general strategy, since the atom being used as a place-holder would have to be guaranteed as falling outside the range of atoms used within the quoted expression itself. However this is a diversion; a much more serious issues has to do with the identity of the pairs and rails used by the constructors into which back-quoted expressions expand. We adopt a policy whereby such expressions expand to a new structure creating expression at the level of the back-quote, and down to and including any level including a comma'ed expression. This is intended as a logical compromise, that simultaneously minimises the chance of unwanted shared tails, but at the same time avoids unnecessary construction. Some examples are given below (note in particular that `[] expands to (RCONS), not to '[]; this is very useful as an abbreviation):

```
`[]                      ≡>  (RCONS)                                    (S4-871)
`[[A B C] [D ,E F]]      ≡>  (RCONS '[A B C] (RCONS 'D E 'F))
`(,FUN 1 ,A 3)           ≡>  (PCONS FUN (RCONS '1 A '3))
`(= (,F A B) (- A B))    ≡>  (PCONS '= (RCONS (PCONS F '[A B]) '(- A B)))
```

Given this machinery, we can then return to the subject of macro procedures, and illustrate some of their properties. Like any other procedures, they can be given own variables, defined embedded in contexts and so forth. The following, for example, is behaviourally equivalent to the INCREMENT$_1$ macro defined above:

```
(DEFINE INCREMENT2                                                      (S4-872)
   (LET [[Y 1]]
      (LAMBDA MACRO [X] `(+ ,X ,↑Y))))
```

This should be contrasted with the following variation, which expands into a form that adds the contextually-relative binding of Y to its argument:

```
(DEFINE ADD-Y                                                           (S4-873)
   (LAMBDA MACRO [X] `(+ ,X Y)))
```

Thus we for example have

```
(LET [[Y 100]] (INCREMENT2 4))    ⇒  6                                   (S4-874)
```

but in contrast:

```
(LET [[Y 100]] (ADD-Y 4))         ⇒  104                                (S4-875)
```

Similarly we have:

```
(LET [[Y 100]] (INCREMENT2 Y))    ⇒  101                                (S4-876)
```

in contrast with

```
(LET [[Y 100]] (ADD-Y Y))         ⇒  200                                (S4-877)
```

Macros can also be recursive, but it turns out on inspection that there are a variety of quite different circumstances all with some vague claim to the phrase "recursive macro". We will distinguish three separate circumstances, only one of which will count as legitimately recursive on our use of that term, but, though coherent, we will suggest that such definitions are probably extremely rare.

The first — and perhaps the most common — sense of the term "recursive macro" describes a definition where the macro translation function yields a structure that may contain uses of its own name. As an example, we will define a multi-argument addition

procedure ++ that will accept any number of arguments. Rather than expanding

```
(++ <A1> <A2> ... <Ak>)                                          (S4-878)
```

into the obvious

```
(+ <A1> (+ <A2> (+ ... (+ <Ak-1> <Ak>)...)))                     (S4-879)
```

our version will instead generate a tree of the following sort:

```
(+ (+ ... (+ <A1> <A2>)                                          (S4-880)
      ... (+ <Ak/2-2> <Ak/2-1>))
   (+ ... (+ <Ak/2> <Ak/2+1>)
      ... (+ <Ak-1> <Ak>)))
```

The definition is as follows (we assume (SUB-RAIL <J> <K> <RAIL>) designates a rail whose elements are the Jth through Kth elements of <RAIL>):

```
(DEFINE ++                                                       (S4-881)
   (LAMBDA MACRO ARGS
      (COND [(EMPTY ARGS) '0]
            [(UNIT ARGS) (1ST ARGS)]
            [$T (LET [[K/2 (/ (LENGTH ARGS) 2)]]
                  `(+ (++ . ,(SUB-RAIL 1 K/2 ARGS))
                      (++ . ,(TAIL K/2 ARGS))))]))))
```

Thus we have the following expansions (we use "≡>" to indicate the lexicalisation of the macro expansion relationship):

```
(++)              ≡>  0                                          (S4-882)
(++ 1)            ≡>  1
(++ 1 2)          ≡>  (+ 1 2)
(++ 1 2 3)        ≡>  (+ 1 (+ 2 3))
(++ 1 2 3 4)      ≡>  (+ (+ 1 2) (+ 3 4))
(++ 1 2 3 4 5)    ≡>  (+ (+ 1 2) (+ 3 (+ 4 5)))
```

and so forth.

What is intuitively "recursive" about this definition is that the structures generated by the first phase — by the *expansion* part of the macro processing — yield structures that may in turn re-invoke the first stage processing when they are processed (in the second stage). Thus the *second-stage* processing of the main macro redex may involve instances of macro redexes defined in terms of the same macro. In such a circumstance the procedure defining the macro — such as the procedure defined in S4-881 — does not involve the *use* of its own name; rather, it *mentions* its own name in the structures it designates. In the example, for instance, the embedded tokens of the name "++" are *quoted*, not used. For

this reason we will call such a macro an ***iterative*** macro, since a) it does not satisfy our definition of self-use in a definition, and b) because the process of macro expansion iterates, but not, so to speak, inside itself; one instance of first-stage macro expansion is over before the next is begun.

Iterative macros are useful and common; our refusal to call them recursive is not intended as a normative judgment. Quite different, however, are what we will call ***recursive macros***: macro procedures whose definition involve a genuine use of the macro in the code that performs the translation function. Not only are they different; they are difficult to motivate. The problem is that it is difficult, given some constraints on macro definitions that are typically obeyed (that we will examine below), to define such a procedure that terminates. As a simple example of a genuinely recursive, but non-terminating, macro, we have the following definition of IF, constructed on the assumption that IF conditional must be discharged into AND and COND expressions (assuming, in other words, that AND and COND were primitive but that IF was not). The design is intended to support uses of IF with either two or three arguments:

```
(DEFINE IF                                                    (S4-883)
   (LAMBDA MACRO ARGS
      (IF (= (LENGTH ARGS) 2)
          `(AND . ,ARGS)
          `(COND [,(1ST ARGS) ,(2ND ARGS)]
                 [$T ,(3RD ARGS)]))))
```

However it is clear that every invocation of IF will cause another invocation of IF, leading to a vicious infinite ascent. It would seem, in order to be useful and well-behaved, that a recursive macro would have to use the macro only in one branch of the definition, which is guaranteed at some point to invoke a different branch of the procedure that did not use the macro name recursively. Though this structure is of course necessarily true of all well-behaved recursive definitions (it has, in other words, nothing special to do with macros), *satisfying* it is much more difficult because, as explained below, a macro is not "supposed" to make decisions based on the *particular* significance of the structure being transformed. The following is such a definition, though without merit:

```
(DEFINE REVERSE-RCONS                                         (S4-884)
   (LAMBDA MACRO ARGS
      (SELECT (LENGTH ARGS)
         [0 '(RCONS)]
         [1 `(RCONS ,(1ST ARGS))]
```

```
[2 `(RCONS ,(2ND ARGS) ,(1ST ARGS))]
[3 (PCONS 'RCONS
          (PREP (3RD ARGS)
                (REVERSE-RCONS (1ST ARGS) (2ND ARGS))))]
[$T (ERROR "Only defined over 0 - 3 arguments")]))))
```

This is well-defined only because the recursive use of REVERSE-RCONS is over two-arguments, which is known to be adequately handled without any such use. Some examples:

```
(REVERSE-RCONS)                          ⇒  '[]                          (S4-885)
(REVERSE-RCONS 'LAPSTREAK)               ⇒  '[LAPSTREAK]
(REVERSE-RCONS 'SHOE 'LEATHER)           ⇒  '[LEATHER SHOE]
(REVERSE-RCONS 'ELEVEN 'TIMES 'SEVEN)    ⇒  '[SEVEN TIMES ELEVEN]
(REVERSE-RCONS '1 '2 '3 '4)              ⇒  <ERROR: Only defined over 0-3>
```

Quite a third kind of informally "recursive" macro is one that employs a recursive procedure to effect the requisite translation. Consider the following definition of a multi-argument addition function, quite different from the ++ of S4-881:

```
(DEFINE +++                                                          (S4-886)
   (LAMBDA MACRO ARGS
      (+++-HELPER ARGS)))

(DEFINE +++-HELPER                                                   (S4-887)
   (LAMBDA EXPR [ARGS]
      (IF (EMPTY ARGS)
          '0
          `(+ ,(1ST ARGS) ,(+++-HELPER (REST ARGS))))))
```

What distinguishes this example is that a recursive procedure is defined whose sole purpose is to create the expanded or transformed structure from the original arguments to the +++ redex. However there is nothing recursive *about* +++ in this case; we merely employ a recursive procedure in defining it. This in fact is perhaps the most common circumstance of the three, but there is no reason to give it a particular name.

What is odd — or at least distinguishing — about the genuinely recursive macro is revealed in terms of the model of it being "meta-level" or suggestively reflective. In the course of the meta-level processing, yet another level shift is employed, leading to yet another dialogue one level above the first one. Recursive macros, in other words, and recursive reflective procedures when we get to them, cause as many reflective shifts of the processor as there are recursive invocations in the course of a given expansion. In 3-LISP each one of these will be run at a different level and with a different environment and continuation structure. With the *iterative* macros, however, the situation is quite different:

the shift back down to regular processing has happened before the second call is encountered; thus an *iterative* macro simply causes a succession of shifts between object level and first meta-level — back and forth as often as necessary. Only two levels are involved, however.

It is sometimes said that macros are procedures that can be *run at compile time.* Though this is hardly a semantically perspicuous remark, we can see in part what is intended by it. Since in 2-LISP the macro body itself has no access to the context in which the expression that it generates will be processed (equivalently, no access to the context in which the original macro redex was processed), the macro cannot itself, by and large, *depend* in any way upon that context. The extreme examples presented in S4-862 and S4-863 show that this convention can be violated, but again in the normal case this is true. Note that, in some sense, this constraint is *more* true than in 1-LISP, where the dynamic context can always be used. It is striking, however, to recognise that it is universally agreed in the standard LISP community that although it *can* be used, it *should not* be used — that such use violates the essential nature of MACROS. As an example, in 1-LISP it is legal (but in bad taste) to define the following:

```
(DEFINE STRANGE1                                ; This is 1-LISP      (S4-888)
   (LAMBDA MACRO (X)
      (IF (EVEN (EVAL X))
          `(+ ,X 1)
          `(- ,X 1))))
```

Thus we would have:

```
(LET ((A 2) (B 3))                              ; These are 1-LISP    (S4-889)
    (STRANGE1 (+ A B))        →   4

(LET ((A 2) (B 4))                                                    (S4-890)
    (STRANGE1 (+ A B))        →   7
```

On the other hand if we construct the following 2-LISP definition:

```
(DEFINE STRANGE2                                ; This is 2-LISP      (S4-891)
   (LAMBDA MACRO [X]
      (IF (EVEN ↓(NORMALISE X))
          (XCONS '+ X '1)
          (XCONS '- X '1))))
```

and try to use it:

```
(LET [[A 2] [B 3]]                                              (S4-892)
   (STRANGE₂ (+ A B))        ⇒  <ERROR: A is unbound>

(LET [[A 2] [B 4]]
   (STRANGE₂ (+ A B))        ⇒  <ERROR: A is unbound>
```

we generate an error, since the call to NORMALISE in the body of STRANGE_2 attempts to normalise the variable A in an environment — the static context of the definition of STRANGE_2 — where it has no meaning.

Note as well that the definition in S4-891 means that the argument to the macro redex will be processed *twice* — thus if it involved side-effects, they would happen twice in the course of complete reduction of the macro redex: once in the first stage of processing, once in the second. It seems unlikely that this is an intended consequence of such a definition.

A full understanding of *why* this is considered ill-formed is best revealed by analysis that goes far beyond that of this dissertation — an analysis where the present dissection of evaluation into normalisation and reference is extended, yielding a conception of normalisation as the *production of a normal-form co-designator of an instance of a schema.* In such a framework a macro can be defined as a *schematic meta-description of a schema*; what examples S4-888 and S4-891 illustrate are *schematic meta-descriptions of instances of schemata.* It is this dependence on the instance that is poorly attempted in 1-LISP, and impossible in 2-LISP. Furthermore, a procedure that is free of dependence on instantiations can of course be compiled because precisely what is available at so-called "compile time" is the schematic structure of a program, but not the instance structure. In fact of course it is IMPRS that are the natural locus of meta-descriptive access to instance structure; that is one aspect of how they most fundamentally differ from MACROS (as well as the more obvious fact that they do not give back to the processor a form to be processed, but merely one to be used directly). Given this analysis, a space of four, rather than two, kinds of procedures begins to emerge, since these two distinctions seem independent and orthogonal. But such talk takes us into areas we are not yet equipped to explore.

In chapter 5 we will discuss macros in greater depth. For present purposes we have two remaining tasks: a) we need to make a comment about parameter matching in macros, and b) we must explain the form of MACRO closures.

Regarding the binding of macro parameters, once again there arises an interaction between our support of non-rail CDRs in general redexes, and the intensional stance — the *non-normalised* manner — in which MACROs receive their arguments for first-stage processing. This is similar to the difficulty we encountered with IF, but in the present case, as we will see, some acceptable solutions can be found.

The trouble is best introduced with an example. We will define a macro called AVERAGE so that expressions of the form:

```
(AVERAGE <E1> <E2>)                                          (S4-893)
```

will be transformed into expressions:

```
(/ (+ <E1> <E2>) 2)                                          (S4-894)
```

Our first definition of AVERAGE is this:

```
(DEFINE AVERAGE1                                             (S4-895)
   (LAMBDA MACRO [E1 E2]
      `(/ (+ ,E1 ,E2) 2))))
```

As expected, we would support the following simple behaviour:

```
(AVERAGE1 10 20)                          ⇒  15              (S4-896)
(LET [[X -5] [Y 5]] (AVERAGE1 X Y))       ⇒  0
```

But a user might be surprised, given that we have:

```
(LET [[Y [10 20]]] (+ . Y))               ⇒  30              (S4-897)
```

to discover the following:

```
(LET [[Y [10 20]]] (AVERAGE1 . Y))  ⇒  <ERROR: Pattern failure>   (S4-898)
```

The problem is that the CDR of the macro redex (AVERAGE$_1$. Y) in S4-898 is of course an atom, not a rail, and therefore there is no way that its designator 'Y can be matched against AVERAGE$_1$'s pattern [E1 E2], even given our rail/sequence extension. Though Y in this context *designates* a sequence, it is not itself a designator that can be piecewise decomposed.

A local solution would be to redefine AVERAGE so as not to require that its argument be decomposable. The following would fail (the problem has merely been shifted):

```
(DEFINE AVERAGE₂                                                    (S4-899)
   (LAMBDA MACRO ARGS
      `(/ (+ ,(1ST ARGS) ,(2ND ARGS)) 2)))
```

Clearly, what is required in this instance is the following:

```
(DEFINE AVERAGE₃                                                    (S4-900)
   (LAMBDA MACRO ARGS
      `(/ (+ . ,ARGS) 2)))
```

However this option is open to us only because we happen to use the exact same sequence as the full argument set to another function. We may not always be so lucky. Consider for example the following seemingly reasonable definition of a function called VOLUME, intended to take three arguments (the X, Y, and Z dimensions of a rectangular solid) and yield the volume. We assume that we have only a two-argument multiplier: thus we propose:

```
(DEFINE VOLUME₁                                                     (S4-901)
   (LAMBDA MACRO [X Y Z]
      `(* ,X (* ,Y ,Z))))
```

(A note in passing: there is no harm in using Z as a formal parameter, even though we are using that name for the circular Y operator of recursion, since within this context no use of that other function is required.) However, although S4-901 will support:

```
(LET [[A 3] [B 5] [C 4]]                                            (S4-902)
   (VOLUME₁ A B C))                    ⇒   60
```

It will as expected fail in this case:

```
(LET [[X [3 5 4]]] (VOLUME₁ . X))    ⇒   <ERROR: Pattern failure>   (S4-903)
```

Nor is any simple solution of the sort employed in S4-900 available, since we cannot use the designator of all three numbers as the argument designator for any interior function. Rather, it would seem that we have to construct an expanded form that explicitly destructures the *referent* of the argument expression, rather than having the macro itself try to decompose the argument expression itself (i.e. *qua* expression). Thus we encounter no problem with:

```
(DEFINE VOLUME₂                                                     (S4-904)
   (LAMBDA MACRO ARGS
      `(* (1ST ,ARGS)
         (* (2ND ,ARGS) (3RD ,ARGS)))))
```

This would still support all of:

```
(VOLUME₂ 3 6 4)                          ⇒  60                    (S4-905)
(LET [[A 1] [B 2] [C 7]]
   (VOLUME₂ A B C))                      ⇒  14
(VOLUME₂ (+ 1 1) (* 1 1) (- 1 1))        ⇒  0
```

and would also properly generate:

```
(LET [[X [3 6 4]]] (VOLUME₂ . X))        ⇒  60                    (S4-906)
```

since ARGS in this last case would be bound to the handle 'X, implying that the VOLUME$_2$ redex would generate as the transformed expression:

```
(* (1ST X) (* (2ND X) (3RD X)))                                    (S4-907)
```

which would clearly normalise to 60.

There is very little that is inspiring about this solution. For one thing, this technique lays down, in its first phase, *three* copies of the argument expression, which must therefore be normalised three independent times, which is an ill design. The processing, for example, of

```
(LET [[X 3]]                                                      (S4-908)
   (VOLUME₂ . (BLOCK (SET X (+ X 1))
                     [X X X])))
```

would return the unlikely result of 120.

Furthermore, there was nothing unique about VOLUME: *every* macro that wished to facilitate the use of objectified arguments would have to use techniques of approximately this sort.

In search of a better solution, we may note that VOLUME merely wanted to *re-arrange* the argument expressions of the redex with which it was called: though the macro did not *itself* normalise the arguments, it constructed an expression in which they *would* be normalised. Thus if instead of S4-904 we defined VOLUME as follows:

```
(DEFINE VOLUME₃                                                   (S4-909)
   (LAMBDA MACRO ARGS
      `(LET [[[X Y Z] ,ARGS]] (* X (* Y Z)))))
```

we would still support all the behaviour in S4-905 and S4-906. Furthermore, this would engender only a single processing of the argument expression, which is happier by far than the previous suggestion. In addition, this technique could be generalised: we could define a procedure called N-MACRO (for "normalising macro"), for those macros that are prepared to normalise all of their arguments independent of the expansion that the macro itself yields.

The idea would be to use N-MACRO *as a type argument to* LAMBDA so as to generate MACRO procedures of the sort illustrated in S4-909. N-MACRO depends for its success on the fact (true in 2-LISP but not in any standard LISP) that the result of processing an expression is an expression that can be processed any number of further times without visible consequence. Thus we would use N-MACRO for example as follows:

```
(DEFINE VOLUME4                                                    (S4-910)
   (LAMBDA N-MACRO [X Y Z] `(* ,X (* ,Y ,Z))))
```

The trouble with N-MACRO, however, is that it solves the problem by essentially avoiding it: MACROS defined in terms of N-MACRO differ from full-fledged EXPRS in no interesting way. To see this, we first note that the body of S4-909 is itself an abbreviation for an expression more complex than, but essentially equivalent to:

```
((LAMBDA EXPR [X Y Z] (* X (* Y Z)))                               (S4-911)
  . <ARGS>)
```

If the definition in S4-909 were converted to an EXPR, rather than an N-MACRO, however, it would look as follows:

```
(DEFINE VOLUME5                                                    (S4-912)
   (LAMBDA EXPR [X Y Z] (* X (* Y Z)))
```

Whether VOLUME is invoked in virtue of its name being bound to an EXPR of S4-912 form, or expands into the equivalent S4-911 form, is surely rather inconsequential. Thus the adoption of N-MACRO does not seem recommended.

(In passing we may discard the situation that the S4-911 proposal is somehow inherently "open-coded" in the sense that is used to discuss compilation strategies: we consider that to be an implementational, rather than a semantic, concern. There is of course no reason that EXPR function definitions can not be used in an "open-coded" form by a compiler.)

Once again, we are forced to conclude that meta-structural machinery (of which MACROS are an example) and the use of non-rail CDRS in object level code to objectify arguments seem rather to collide (perhaps suggesting, as this author believes but counter our entire approach here, that objectification may be *inherently* meta-structural in some deep sense). As a pragmatic, if not elegant, solution, we can adopt the tactic that we employed in defining IF, in conjunction with the fact that the normalisation of an expression can be used in place of the expression itself. We propose, in other words, a

variant of N-MACRO, to be called S-MACRO, that binds the pattern to the argument expression as a rail *if possible*, and if not (i.e., if the argument expression is *not* a rail), engenders the normalisation of that expression prior to the matching. This visible or behavioural consequence of this strategy, as with IF, will be only this: when a non-rail CDR is used, then that CDR will be normalised in its entirety, and at the beginning of the translation stage of macro reduction, even if the macro would on rail CDRs normalise the argument expressions only selectively. We expect in the definition of S-MACRO, in other words, to encounter something like the following code:

```
(BIND PATTERN                                                    (S4-913)
      (COND [(ATOM PATTERN) ARGS]
            [(RAIL PATTERN)
             (IF (RAIL ARGS)
                 ARGS
                 (NORMALISE ARGS))]
            [$T (ERROR "Illegal pattern structure")]))
```

The problem, however, is that in order to use this code correctly, the call to NORMALISE in the penultimate line needs to pass as an explicit argument the environment in force when the macro redex is itself processed, or else it needs to cause the macro itself to expand into an explicit binding operation, of the sort pursued in the definition of N-MACRO. How such an S-MACRO would differ from N-MACRO is that it would expand into code of the sort illustrated in S4-909 *only if necessary*, whereas N-MACRO generated this kind of "wrapping" code in all circumstances.

The strategy here is only partially satisfactory, which recommends against adopting it in our primitive definition of the MACRO facility in 2-LISP. *Defining* S-MACRO, however, is beyond the realm of possibility in this dialect, because of the ensuing context complexities. Once again, therefore, we will back off in our attempt, deferring our final solutions until 3-LISP. In that dialect defining S-MACRO (and indeed any number of other types of MACRO functions) will be readily possible: thus the dialect itself will not have to make a decision. For the time being, therefore, we will content ourselves with the simpler macros of S4-901 style, which either *require* that the argument expressions be rails, or else do their destructuring explicitly (as exemplified in the definition of $AVERAGE_3$). An unhappy conclusion, but one we will fortunately soon be able to relinquish.

In part by way of review, and in part in order to illustrate another often useful technique for coping with these problems, we will as a last example define a macro version

of AND as suggested in section 4.b.ix, where we indicated that we would like expressions of the form

```
(AND <S1> <S2> ... <Sk>)                                        (S4-914)
```

to expand into

```
(IF <S1>                                                        (S4-915)
    (IF <S2>
        (IF <S3>
            ... (IF <Sk> $T $F) ...
            $F)
        $F)
    $F)
```

A simple definition of AND using a recursive meta-level procedure is the following:

```
(DEFINE AND1                                                    (S4-916)
   (LAMBDA MACRO ARGS (AND* ARGS)))

(DEFINE AND*                                                    (S4-917)
   (LAMBDA EXPR [ARGS]
      (COND [(EMPTY ARGS) '$T]
            [(UNIT ARGS) (1ST ARGS)]
            [$T `(IF ,(1ST ARGS)
                     ,(AND* (REST ARGS))
                     $F)])))
```

However this will not support non-rail CDRs. We suggested in that section the following definition, which we can now explain.

```
(DEFINE AND2                                                    (S4-918)
   (LAMBDA MACRO ARGS
      (IF (RAIL ARGS)
          (AND* ARGS)
          `↓(AND* ↑,ARGS))))
```

Thus we have the following expansions:

```
(AND2)                      ≡>  $T                              (S4-919)
(AND2 (ATOM 'X))            ≡>  (ATOM 'X)
(AND2 (= 1 1) (= 1 2))      ≡>  (IF (= 1 1) (= 1 2) $F)
(AND2 A B C D E)            ≡>  (IF A (IF B (IF C (IF D E) $F) $F) $F)
(AND2 . (REST [X Y Z]))     ≡>  ↓(AND* ↑(REST [X Y Z]))
```

We can now see how this works. The basic idea is to recognise that the macro itself is defined in terms of a subsidiary but meta-level procedure that takes an expression — crucially decomposable — and constructs from it the appropriate conditional expression. What the main macro then does is this: *if* the expression can be decomposed in general, it does that and returns the appropriate conditional straight away. If it cannot (the CDR of the

macro redex is not a rail) then it returns an expression that says *first* to normalise the CDR, and then to have the meta-level procedure construct the appropriate conditional *from that normalised* CDR, which is *guaranteed* to be a rail. In other words although the macro cannot *itself* process the CDR (since it has no access to the appropriate context), it can yield an expression that will cause that processing to happen. The term ↑.ARGS will normalise and obtain a handle of the result, in the appropriate context, as we have already seen; the term ↓(AND* ↑.ARGS) will yield, as the result of the processing, the appropriate term to dereference (thereby processing) the conditional that is subsequently constructed.

It should be noted that this works because of the idempotence of Ψ: the expression that AND* will create will be built out of the *results* of normalising the CDR, rather than out of the un-normalised CDR in the standard case. However this is of course perfectly acceptable.

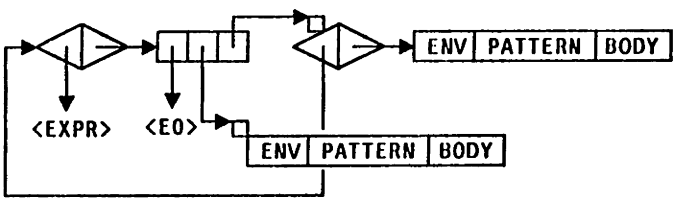
Though there is perhaps a certain ingenuity to this technique, it can hardly be called elegant. A better solution will be obtained in chapter 5, where macro procedures will, if necessary, be able to perform their own normalisations.

Two final tasks must be discharged. First we need to establish the structure of the primitive MACRO closure. Like IMPR and EXPR, MACRO will in the initial environment be bound to a primitive and circular closure of no particular content, other than betraying its (type) stability. In our extended lexical notation it would be printed as follows:

```
MACRO   ⇒   M: (<EXPR> E0                                    (S4-920)
                       '[ENV PATTERN BODY]
                       '(:M ENV PATTERN BODY))
```

This was in fact indicated in S4-731. Graphically, <MACRO> would be notated in this way:

(S4-921)



Finally, we need to inquire about the semantics of this primitive. The crucial fact about macro procedures is this: *the function designated by the body of a macro closure is the function computed in the first stage of processing*; the second stage follows in virtue of the surrounding context and the structure yielded as the result of that processing. In other

words, if M is a macro procedure whose normal-form is (<MACRO> <E1> <P1> <F1>), then the significance of an M redex — an expression of the form (M . <ARGS>) will be the significance, very approximately, of (Φ(F1) . ARGS). It is this fact which we should most expect to be revealed by the semantical account.

As we saw in section 4.c.iv, there are *three* things that we need to explicate: the general significance of <MACRO>, the internalised function engendered by <MACRO>, and the significance of non-primitive MACRO redexes (this is *all* MACRO redexes, as it happens, since no MACROs are primitive — <MACRO> itself, like <IMPR>, is an EXPR). The first two are relatively simple:

```
Σ[E0("MACRO)]                                                    (S4-922)
   = λE.λF.λC .
       C("(E0("EXPR) E0 [ENV PATTERN BODY] '(<MACRO> ENV PARAM BODY)),
         [λEc.λSp.λSb .
             [λ<S1,E1,F1> .
                 Σ(Sb,
                   EXTEND(Ec,Sp,HANDLE(S1)),
                   F1,
                   [λ<S2,D2,E2,F2> .
                         Σ(D2,E1,F2,[λ<S3,D3,E3,F3> . D3])])]],
         E,F)
```

We can see in the designation of MACRO the essential properties beginning to emerge. In particular, note how a MACRO closure, when reduced with a structure S_1 in a context E_1 and F_1, obtains the designation D_2 of its own body expression (in the context appropriately extended by binding its pattern to a *designator* of the arguments: in this way MACROs are just like IMPRS), and then *obtains the significance of that designation* in the appropriate context (the E_1 that the macro redex was processed in, and the field that as usual has been passed straight through).

The internalisation of MACRO is essentially identical to that of EXPR (in S4-626) and IMPR (in S4-628): it reveals only that MACRO redexes whose arguments are in normal-form are stable:

```
Δ[E0("MACRO)]                                                    (S4-923)
   = λS.λE.λF.λC .
       Σ(NTH(1,S,F),E,F,
         [λ<S1,D1,E1,F1> .
             Σ(NTH(2,S,F1),E1,F1,
               [λ<S2,D2,E2,F2> .
                   Σ(NTH(3,S,F2),E2,F2,
                     [λ<S3,D3,E3,F3> .
                         C("(E0("MACRO) S1 S2 S3),E3,F3)])])])
```

But of some interest is the following: the internalisation of non-primitive MACRO closures (for it is this that will manifest the computational consequences of MACROs when they appear in reductions):

$$\forall S_3, S_p, S_b \in \mathcal{S},\ E \in \mathit{ENVS} \qquad \text{(S4-024)}$$
$$[\![\, S_e = \mathrm{ENV}(E) \,]\!] \supset$$
$$[\, \Delta\Gamma\text{"}(\underline{E_0(\text{"}\mathit{MACRO}\text{)}}\ \underline{S_e}\ \underline{\mathrm{HANDLE}(S_p)}\ \underline{\mathrm{HANDLE}(S_b)})\urcorner$$
$$= \lambda S_1.\lambda E_1.\lambda F_1.\lambda C_1\ .$$
$$[\Sigma(S_b, E^*, F_1,$$
$$[\lambda\langle S_2, D_2, E_2, F_2\rangle\ .\ [\Sigma(\mathrm{HANDLE}^{-1}(S_2), E_1, F_2, C_1)]]\,)]\,]\!]$$
where E^* *is like* E_1 *except extended*
by matching $\mathrm{HANDLE}(S_1)$ *against* S_p.

If a non-primitive macro is reduced with arguments S_1, in other words, the body S_b of the MACRO will be processed in an environment E^* which comes from matching the *designator* of the arguments to the MACRO's pattern S_p. This processing must return a handle (a structure designator) S; *the significance of the MACRO redex is the significance of this generated structure* in the original environment E_1 and the current field F_2.

Perhaps the most interesting fact about this last equation is the striking similarity it will bear to the definition of MACRO as a user 3-LISP procedure in chapter 5. We are slowly getting to the point where the meta-theoretic and procedural definitions can be seen in very close parallel.

4.d.vi. The Normalisation ("Flat") and Type Theorems

2-LISP has been completely described; it would therefore be appropriate at this point to prove the two theorems about it that are of most interest to us: the *normalisation* (or *flat*) theorem, stating that designation is preserved by the processor, and the *semantical type* theorem, stating that any designator of each of the five semantical categories is mapped by the processor onto an element of that category's corresponding syntactic type. We have admitted, however, that we will not actually demonstrate proofs of these theorems, though at various points along the way (such as in the proof that CAR was *standard* in section 4.b.ii) we have shown how and why the results are true.

What we will in this section do, however, is to state the two theorems precisely, and bring to light various subtleties that have so far been overlooked. We start with the normalisation theorem. In S3-4 we gave our simplest formulation of this property:

$$\forall S \in \mathbf{S}\ [[\Phi(S) = \Phi(\Psi(S))] \wedge [\text{NORMAL-FORM}(\Psi(S))]] \qquad \text{(S4-928)}$$

Then in S3-132 we gave a more complete context-relative version, as follows:

$$\forall S \in \mathbf{S},\ E \in \mathbf{ENVS},\ F \in \mathbf{FIELDS} \qquad \text{(S4-929)}$$
$$[[\Phi EF(S) = \Phi EF(\Psi EF(S))] \wedge [\text{NORMAL-FORM}(\Psi EF(S))]]$$

This was based on the following definitions of Ψ and Φ in terms of Σ (these are from S3-130):

$$\Psi \equiv \lambda E.\lambda F.\lambda S\ .\ [\Sigma(S,E,F,[\lambda X\ .\ X^1])] \qquad \text{(S4-930)}$$
$$\Phi \equiv \lambda E.\lambda F.\lambda S\ .\ [\Sigma(S,E,F,[\lambda X\ .\ X^2])]$$

Another particularly simple expression of the same theorem, using the NFD predicate defined in S4-100, is this:

$$\forall S \in \mathbf{S},\ E \in \mathbf{ENVS},\ F \in \mathbf{FIELDS}\ [\text{NFD}(\Psi EF(S), \Phi EF(S))] \qquad \text{(S4-931)}$$

Rather than simplifying the presentation of the result, however, it is instructive to recast the main theorem by discharging all of these abbreviations; as the discussion in section 4.b.ii intimated, we thereby encounter some subtleties about contexts that need attention. In particular, we get the following by straightfoward substitution:

$$\forall S \in \mathbf{S},\ E \in \mathbf{ENVS},\ F \in \mathbf{FIELDS} \qquad \text{(S4-932)}$$
$$[[\Sigma(S,E,F,[\lambda X\ .\ X^2]) = \Sigma(\Sigma(S,E,F,[\lambda X\ .\ X^1]),E,F,[\lambda X\ .\ X^2])] \wedge$$
$$[\text{NOPMAL-FORM}(\Sigma(S,E,F,[\lambda X\ .\ X^1]))]]$$

However we can convert this into the following more perspicuous form:

```
∀S ∈ S, E ∈ ENVS, F ∈ FIELDS                                        (S4-933)
  [[ Σ(S,E,F,
      [λ<S1,D1,E1,F1> .
          Σ(S1,E1,F1,
            [λ<S2,D2,E2,F2> . [[D1 = D2] ∧ [NORMAL-FORM(S1)]]])]) ]]
```

This is much more similar in structure to the standard continuation-passing meta-theoretic characterisations we are familiar with, from our many examples, although in this particular case the "continuation" is a predicate, rather than a function, with the consequence that the overall expression is a sentence, rather than a term (as of course we expect).

It looks superficially as if there might be a problem with this: what this says is that the *result* of normalising a term will have the same designation *in the resulting context* (E_1 and F_1) as S did in the *original* context. A moment's thought, however, makes us realise that this doesn't matter, since normal-form designators are context-independent designators *by definition* (we assume the S3-191 definition of "NORMAL-FORM" throughout). Thus if the second part of the predicate (NORMAL-FORM(S_1)) is true, it doesn't matter what context we pass to determine the designation of S_1, since it won't depend on it. Therefore, according to this revised understanding, S4-933 should be provably equivalent to:

```
∀S ∈ S, E ∈ ENVS, F ∈ FIELDS                                        (S4-934)
  [[ Σ(S,E,F,
      [λ<S1,D1,E1,F1> .
          Σ(S1,E,F,
            [λ<S2,D2,E2,F2> . [[D1 = D2] ∧ [NORMAL-FORM(S1)]]])]) ]]
```

However we in fact do have a minor problem; the original worry indeed has some merit, suggesting S4-933 to be the more proper formulation. The difficulty has to do with the same thing that has caused us difficulty all along: the encoding of environments within closures. We have not made our meta-theoretic account honest to our claim that environments and environment-designating rails will somehow (magically) be kept synchronised — a change to one being reflected instantaneously in a change to the other. In point of fact closures are *not* strictly environment independent in their designation: if the field changes in such a way that their enclosed environment designators are modified, they will then designate different functions. We even *advertised* this as a feature, in section 4.c.vi, when we discussed the side effects engendered by the use of SET and DEFINE. And, once put, it is clear we cannot have both things: context independent function designators,

and the ability, by altering the field, to change the designation of previously constructed normal form designators. The two *desiderata* are in outright contradiction.

This is not a difficulty we can take up formally, without first facing the task of aligning meta-theoretic environments and structurally encoded environment designators. However some directions can be indicated. First, it would be possible to identify a normal-form structure not as a single structural field element (like a pair or a rail) but rather as a composite object, containing all of the structures locally accessible from it (i.e., using an environment-free version of ACCESSIBLE* that did not follow bindings). This was what in section 4.a.ix we did in defining a *redex*. A closure, then, would *include as part of its very self* the environment designator, on such a view. Once this move had been taken, we could redefine NORMAL-FORM to be true *of a composite structure* that was context independent *providing it itself was not altered.*

Once this suggestion is raised, we can see that such a move is in fact required in any case, because of rails. We cannot in honesty claim that the rail [1 2 3] (we will call this rail γ) designates the sequence of the first three natural numbers *independent of context*, for in some other context the rail γ might have its tail chopped off, or modified, so that γ became [1 2] or [1 2 (Z . Z)]. Now of course these are gratuitous challenges, for the point we had in mind was that *so long as* γ *exists*, it designates the sequence in question. But the point is that closures' dependence on internal environment designators, and this problem with rails, could be solved with the same machinery.

We should return to closures. If we define a closure to be a *redex*, rather than a *pair*, in other words, and complicate our definitions of "context-independent" to mean "independent of context so long as the designator's identity is preserved", then our original results stand. Thus we can assume the intent and even the formulations of S4-928 through 933 after all — what must be changed is merely the definition of *S* to include composite as well as atomic elements, and the definition of NORMAL-FORM.

We will not pursue the details of these manoeuvres here, but before leaving the topic, it is well to point out, with some simple examples, some things that the normalisation theorem does *not* say. In particular, from the fact that any redex of the form

(<PROCEDURE> . <ARGS>) (S4-935)

will return an expression s that is guaranteed to designate what S4-936 designates, nothing whatsoever is implied as to the relationship between the designation of s *and the designation of* <ARGS>. We raise this because there is a natural tendency to think that our inclusion of primitive "referencing" and "de-referencing" functions (NAME and REFERENT) in the calculus entails that we cannot have a designation-preserving processor. However these are entirely unrelated matters. In fact our whole reason for including NAME and REFERENT was to facilitate the crossing of levels (the obtaining of access to entities at different semantic levels) *within the confines of a semantically flat formalism*, since in very virtue of the normalisation theorem the processor itself would not give us this power. Consider for example the expression

```
(NAME 3)                                                    (S4-936)
```

which normalises to the handle '3. The numeral 3 — the *argument* to NAME — designates of course the third natural number; the result of normalising S4-936 designates a *numeral*; the number and the numeral are, it cannot be denied, absolutely different. What the semantically flat processor guarantees is that *the entire expression S4-936* and its result are co-designating, and this is of course true: the redex (NAME 3) and the handle '3 both designate the numeral.

In standard situations there is of course no tendency to think that the mere ability to use functions challenges semantic flatness. Thus from the fact that

```
(INCREMENT 3)                                               (S4-937)
```

simplifies to 4 no one would suspect that 2-LISP's Ψ is a function that takes terms X onto terms Y that designate the successor of the designation of X. It is *INCREMENT* that designates the successor function, and INCREMENT is by no means designation preserving. Our claim is only that *NORMALISE* is designation preserving. Though this is transparent in the arithmetic case, the use of explicitly semantical functions within the dialect — NAME and REFERENT, in other words — is apparently liable to engender confusion. To reiterate, neither NAME nor REFERENT is a designation-preserving function: it is in fact exactly because they are *not* designation preserving that they are useful. Rather, it is only NORMALISE — 2-LISP's Ψ — that is advertised, and relied on, as having this property.

Another potential confusion to defuse has to do with the relationship between 2-LISP's designation-preserving Ψ and any claim that 2-LISP's Ψ is purely *extensional*. In our

use of this last predicate, we have always been careful to state that a context of occurence of a term is extensional if the *designation* of the overall composite structure depended only on the designation of the constituent. In this sense much of 2-LISP is extensional, including Ψ — NORMALISE (the "input" to NORMALISE can be viewed as an "ingredient" expression since if [S_1 ⇒ S_2], then equivalently [S_2 = NORMALISE(S_1)], and in the latter formulation the whole is the result; the ingredient is the input). However this does not imply that the *intensional* properties of the results of normalising an expression may not depend in various ways on the intensional properties of the input.

Perhaps the most complex example of this intensional dependency arose in our consideration of LAMBDA, but surely the *simplest* instances have to do with what we called the *stability* of normal-form designators. From the fact that normal-form expressions normalise to themselves, plus the fact that not all normal-form designators are canonical, it follows that normalisation is not blind to the intensional properties of its inputs. Intensional properties may be "passed through" Ψ, in other words, in various at times complex ways, all within the *extensional* constraints demanded by the designation-preserving aspects of the main processor function.

As opposed to the normalisation theorem, we have not previously put the *semantical type theorem* into formal language. Briefly, its claim is this: all expressions that *designate* elements of each semantical category (numbers, truth-values, sequences, s-expressions, and functions) will *normalise* to an expression of a given *structural* category (numerals, booleans, rails, handles, and closures, respectively). Though we defined a meta-theoretic TYPE function in S4-244, it is simplest to state the theorem directly in terms of category membership. The first proposal is this (note that we ease up a little, by restricting its application to expressions that do in fact return a result):

```
∀S1,S2 ∈ S, E ∈ ENVS, F ∈ FIELDS, D ∈ D                          (S4-938)
    [[[ S2 = ΨEF(S1) ] ∧ [ D = ΦEF(S1) ]] ⊃
    [ if [D ∈ S] then [S2 ∈ HANDLES]
      elseif [D ∈ INTEGERS] then [S2 ∈ NUMERALS]
      elseif [D ∈ TRUTH-VALUES] then [S2 ∈ BOOLEANS]
      elseif [D ∈ SEQUENCES] then [S2 ∈ RAILS]
      elseif [D ∈ FUNCTIONS] then [S2 ∈ CLOSURES]]]
```

Given the definition of D in S4-143, this covers all possibilities except for designators of entities in the user's world, about which we have nothing to say.

There is, however, one term in S4-938 that we have not previously defined: ***CLOSURES***. Our intent is clear; the definition is not. The obvious suggestion is this:

$$\textit{CLOSURES} \equiv \{P \in \textit{PAIRS} \mid\mid [CAR(P,F) \in \{E_0("\textit{IMPR}), E_0("\textit{EXPR}), E_0("\textit{MACRO})\}]\} \quad \text{(S4-939)}$$

The problem, however, is that F is undefined in this term. We could make ***CLOSURES*** a function of F, but the appropriate F (the one *resulting* from the normalisation indicated in the second line of S4-938) is not explicitly available. We could reformulate S4-938 so as to change this fact, but since the NORMAL-FORM theorem guarantees that the S_2 of S4-938 will be a closure if it is a pair, it is simpler merely to change this theorem to claim only that function designators will be pairs. Thus we have instead (this will stand as our official statement of the theorem):

$$\begin{aligned}&\forall S_1, S_2 \in \textit{S}, E \in \textit{ENVS}, F \in \textit{FIELDS}, D \in \textit{D} \quad \text{(S4-940)}\\ &\quad [\![[S_2 = \Psi EF(S_1)] \wedge [D = \Phi EF(S_1)]] \supset\\ &\quad [\textit{if } [D \in \textit{S}] \textit{ then } [S_2 \in \textit{HANDLES}]\\ &\qquad \textit{elseif } [D \in \textit{INTEGERS}] \textit{ then } [S_2 \in \textit{NUMERALS}]\\ &\qquad \textit{elseif } [D \in \textit{TRUTH-VALUES}] \textit{ then } [S_2 \in \textit{BOOLEANS}]\\ &\qquad \textit{elseif } [D \in \textit{SEQUENCES}] \textit{ then } [S_2 \in \textit{RAILS}]\\ &\qquad \textit{elseif } [D \in \textit{FUNCTIONS}] \textit{ then } [S_2 \in \textit{PAIRS}]]]\end{aligned}$$

Again, we will offer no proof of this theorem; it is unlikely, however, that the reader will not have come to believe it in virtue of our many examples.

4.d.vii. The 2-LISP Meta-Circular Processor

A great many of the procedural and computational aspects of 2-LISP are revealed in the 2-LISP meta-circular processor. In this section we will present and briefly explain the code for one version of such an abstract machine. Four things, however, should be noted before we begin. First, *no inkling of the declarative semantics is revealed by this code* — all that the meta-circular processor manifests is procedural consequence. It is for this reason that we have deferred the discussion of this processor until late in the chapter, since it has been primarily with declarative issues that we have been concerned. A quick glance at the code in the following pages will show how much simpler is this purely procedural account than the meta-theoretic characterisations of full semantics we have explored in the last four sections. Secondly, there are any number of ways in which a meta-circular processor can be constructed, as discussed in chapter 2. We will focus here on a tail-recursive continuation-passing version, since it maximally encodes the state of the computation being executed in explicit argument structures, rather than in the state of the meta-circular processor itself. Third, the semantic flatness of 2-LISP's Ψ is in part reflected in the absence, in this code, of up-arrows and down-arrows. There are a few notable exceptions (such as in the case of simple primitives), but by and large the terms in the meta-circular processor *designate* the terms being processed; the maintenance of a clear separation between semantic levels is relatively strict. Fourth and finally, the comparative simplicity of the meta-circular processor in part stands witness to our success in developing an elegant dialect. To the extent that a formalism is simple, its internal self-description is typically doubly simple; to the extent that it is complex, the self-description is doubly complex — for the attempt to describe a baroque language in a baroque language can unleash multiplicative confusion.

It will be useful for the reader to obtain a relatively thorough understanding of this code at this point, since the 3-LISP reflective processor, which will be a central part of the definition of 3-LISP, is based on this 2-LISP meta-circular processor, but modified in ways that make it difficult to understand without a prior grasp of the simple case. One further comment: for simplicity, we will not attempt to include explicit error-checking in the code; we will always assume that the structures being processed are both syntactically and semantically well-formed.

We begin with the NORMALISE — the main processor function:

```
(DEFINE NORMALISE                                                    (S4-946)
   (LAMBDA EXPR [EXP ENV CONT]
      (COND [(NORMAL EXP) (CONT EXP)]
            [(ATOM EXP) (CONT (BINDING EXP ENV))]
            [(RAIL EXP) (NORMALISE-RAIL EXP ENV CONT)]
            [(PAIR EXP) (REDUCE (CAR EXP) (CDR EXP) ENV CONT)])))
```

If an expression is in normal-form, it is sent to the continuation unchanged; thus the first clause in the conditional will catch all the handles, booleans, and numerals. Thus we have the following basic category structure to the NORMALISE dispatch: the first three of the six structural categories are treated in the first clause, and the other three are discharged in the remaining three clauses. However there is one exception to this simple characterisation: *normal-form* pairs (closures) and *normal-form* rails are recognised by NORMAL (a predicate we will define below), and are sent directly to the continuation; this is so that the *exact structure* can in each case be returned (ensuring that ↓ be *token*-idempotent). If this were not done, and they were dispatched in the subsequent clauses of the conditional like other members of their structural category, a *type*-equivalent structure would in each case be returned, but it would not be the same one.

NORMALISE and REDUCE form a mutually-recursive pair; the definition of the latter is as follows:

```
(DEFINE REDUCE                                                       (S4-946)
   (LAMBDA EXPR [PROC ARGS ENV CONT]
      (NORMALISE PROC ENV
         (LAMBDA EXPR [PROC!]
            (SELECTQ (PROCEDURE-TYPE PROC!)
               [IMPR (IF (PRIMITIVE PROC!)
                         (REDUCE-IMPR PROC! ↑ARGS ENV CONT)
                         (EXPAND-CLOSURE PROC! ↑ARGS CONT))]
               [EXPR (NORMALISE ARGS ENV
                        (LAMBDA EXPR [ARGS!]
                           (IF (PRIMITIVE PROC!)
                               (REDUCE-EXPR PROC! ARGS! ENV CONT)
                               (EXPAND-CLOSURE PROC! ARGS! CONT))))]
               [MACRO (EXPAND-CLOSURE PROC! ↑ARGS
                         (LAMBDA EXPR [RESULT]
                            (NORMALISE RESULT ENV CONT)))])))))
```

We will generally adopt a convention of using, for parameters that designate normal-form structures, a name formed by adding an exclamation point to the name used for the un-normalised structure. Thus REDUCE, given a procedure, arguments, and an environment and continuation, first normalises the procedure, with a continuation that binds the result to the

parameter PROC!. It is not that PROC! is itself in normal form (no atom is in normal form) nor that PROC! is *bound* to a normal-form structure (all atoms are bound to normal form structures); rather, PROC! *designates* a normal-form expression. That PROC is normalised like any other argument reflects the fact that 2-LISP is a higher order dialect.

It is very important to note that each call to NORMALISE throughout the entire meta-circular processor is *tail recursive*; thus if this code were itself being run by a continuation-passing processor, the NORMALISE redex in the third line would be given the same continuation as the REDUCE redex that originally caused this code to be run. We will review this fact more carefully in chapter five, where it will matter more crucially, but we will honour this aesthetic throughout the present code, by way of preparation.

Once the normalised procedure PROC! has been received, REDUCE dispatches on its type. Note that we cannot use TYPE, since it is a *procedure* type we want to select on, not a *referent* type (the term (TYPE PROC!) would always return 'PAIR, since PROC! will always designate a closure, and (TYPE ↓PROC!) would always return 'FUNCTION, for the same reason). If PROC! designates the EXPR closure, the redex arguments are in turn normalised (again tail-recursively); it if designates the IMPR closure, they are not. In either case, a check is made to see whether the closure is one of those primitively recognised. If so, the reduction is treated primitively (below); if not, a general EXPAND-CLOSURE is called to bind the closure pattern to the arguments and normalise the closure body.

If the expression is a MACRO redex, no primitive check need be made since there are no primitives MACROS. Instead EXPAND-CLOSURE is run *as the first phase of macro reduction*; the result that it returns is then handed right back to NORMALISE, *as the second stage*. Note that the second stage call to NORMALISE is tail recursive in two senses: it honours our aesthetic that all syntactic recursion within the processor be intensionally iterative, but in addition NORMALISE is called with the same ENV and CONT that were originally used to process the MACRO redex. Once the MACRO has generated the appropriate structure for second stage normalising, in other words, its job is done; no name for that part of the processing needs to be retained in the second stage.

EXPAND-CLOSURE normalises the body of the closure in an environment formed by extending the environment extracted from the closure itself with the appropriate bindings caused by matching the arguments against the closure pattern (PATTERN, BODY, and ENV,

defined below, are straightforward utility selector functions on closures). Note that EXPAND-CLOSURE is not called with the environment from the context of the original redex; thus we see the embodiment of the static scoping protocol:

```
(DEFINE EXPAND-CLOSURE                                              (S4-947)
   (LAMBDA EXPR [CLOSURE ARGS CONT]
      (NORMALISE (BODY CLOSURE)
                 (BIND (PATTERN CLOSURE) ARGS (ENV CLOSURE))
                 CONT)))
```

In addition, we see that the body is normalised with the original continuation passed down from NORMALISE through REDUCE. In other words, as Steele and Sussman have pointed out,[10] embedding of continuations has fundamentally to do with recursively normalising *arguments*, not with *reducing procedures*. The calls to NORMALISE on the third and ninth lines of S4-946 employ what we will call ***embedding*** continuations: continuations that contain within themselves a binding to a simpler continuation; the present NORMALISE, however, does not.

Next we look at the reduction of primitives; sorted into primitive IMPRS and primitive EXPRS. There are in 2-LISP three primitive IMPRS: SET, LAMBDA, and IF. SET normalises its *second* argument, re-binding the un-normalised first argument to the normal-form expression that results in the current environment. We will define REBIND below, but it was also introduced earlier in section 4.d.iv. Note the semantic flatness of SET that we commented on before: although a SET redex is intensional in first position, the parameter there is at the same level as the second normalised argument. In the present code, therefore, (1ST ARGS) will *designate* the parameter — i.e., if the governing redex was (SET X (+ 2 3)) then (1ST ARGS) would designate X and (2ND ARGS) would designate (+ 2 3). Therefore in the call to REBIND, (1ST ARGS) would *normalise* to the handle 'X, and BINDING would normalise to the handle '5. Thus we need no complicated level-shifting primitives to keep things straight.

```
(DEFINE REDUCE-IMPR                                                 (S4-948)
   (LAMBDA EXPR [PROCEDURE ARGS ENV CONT]
     (SELECT ↓PROCEDURE
        [SET (NORMALISE (2ND ARGS) ENV
                 (LAMBDA EXPR [BINDING!]
                    (CONT (REBIND (1ST ARGS) BINDING! ENV))))]
        [LAMBDA (NORMALISE (1ST ARGS) ENV
                    (LAMBDA EXPR [CLOSURE-TYPE]
                       (REDUCE CLOSURE-TYPE
                               ↑[ENV (2ND ARGS) (3RD ARGS)]
```

```
                                ENV CONT)]
    [IF (NORMALISE (1ST ARGS) ENV
            (LAMBDA EXPR [FIRST!]
               (IF (= FIRST! '$T)
                   (NORMALISE (2ND ARGS) ENV CONT)
                   (NORMALISE (3RD ARGS) ENV CONT))))]))))
```

Note that the SELECT (not SELECTQ) uses ↓PROCEDURE, rather that PROCEDURE, because it uses actual equality with the primitive closures of the implementing language to determine identity. This interacts with the form of the initial environment, which we will discuss below.

LAMBDA, as we mentioned earlier, passes the buck to its first argument, after first normalising it. Although this will typically be one of the primitive EXPR, IMPR, or MACRO closures, it may of course in general be an arbitrary user procedure. What is important about this treatment is the level shift we encountered earlier: that procedure that is to establish the closure is given *designators* of the environment, pattern, and body expressions. Since REDUCE requires a designator of the argument structure, we explicitly construct the appropriate rail. The actual environment and continuation in which this reduction happens, however, remain meta-level theoretical posits; hence the third and fourth arguments to REDUCE are normal; it is only the second argument that involves the passing down of structures from this level.

Finally we have the conditional. First the premise (the first argument) is normalised; upon return its result is checked. We of course have to see whether the *boolean* '$T is returned (again because of semantic flatness). Again all calls to NORMALISE in our code are tail-recursive; in addition, the second or third argument to IF — the appropriately selected consequent, in other words — is normalised with the same continuation as was the original IF redex (i.e., these continuations are not embedding). For example, if a conditional redex R is of the form (IF <P> <E1> <E2>), the normalisation of either <E1> or <E2> will be given the same continuation as R was given originally.

There are about two dozen primitive EXPRs that we need to treat as well. Three of these, as the discussion in the last few sections has indicated, involve a subsequent normalisation in the current context: REFERENT, NORMALISE, and REDUCE. It is for this reason that REDUCE-EXPR must be given the environment and continuation as explicit arguments. In the following code these three are treated first:

```
(DEFINE REDUCE-EXPR                                                    (S4-949)
   (LAMBDA EXPR [PROC! ARGS ENV CONT]
      (SELECT PROC!
         [↑REFERENT   (NORMALISE ↓(1ST ARGS) ENV CONT)]
         [↑NORMALISE (NORMALISE ↓(1ST ARGS) ENV
                        (LAMBDA EXPR [RESULT] (CONT ↑RESULT)))]
         [↑REDUCE     (REDUCE ↓(1ST ARGS) ↓(2ND ARGS) ENV
                        (LAMBDA EXPR [RESULT] (CONT ↑RESULT)))]
         [$T (CONT ↑(↓PROC! . ↓ARGS))])))
```

REFERENT first de-references its own argument (which must be a handle, since it must be a normal-form structure designator), and then normalises the result. Note that the original continuation is passed along to that normalisation: thus the second normalisation mandated by a REFERENT redex is given the same continuation as the original. From the level-shift here it is clear that REFERENT is fundamentally a level-crossing procedure. Both NORMALISE and REDUCE, however, embed a continuation within the recursive calls to NORMALISE and REDUCE ensuring that the original continuation is given a *designator* of the result, rather than the result itself. This is of course necessary. We see as well how the same environment is used for the embedded normalisation in both cases: ENV is given as the penultimate argument to the recursive calls to NORMALISE and REDUCE. In a more adequate dialect this argument would be (3RD ARGS) (not ↓(3RD ARGS), since environments are not object level structures!).

There are two options regarding the other 25 primitives. Since this is a meta-circular processor — since, in particular, the structures we are processing are *structurally* identical to those in the language we are using — we can simply complete the definition of REDUCE-EXPR as indicated in S4-949. This works because none of those 25 involve any explicit access to the environment or continuation, which we would otherwise have to pass to them explicitly. This can be put another way: *if* a primitive does involve environment or continuation, then the last line in S4-949 would fail, for it would cause the interaction to be with the environment and continuation structures *of the meta-circular processor itself*, rather than with the explicit environment and continuation structures *that the meta-circular processor mentions*. If, for example, we treated SET redexes in this fashion (say, (SET X 3)), then X would be set to 3 in the *meta-circular processor's* environment, not in the environment that the meta-circular processor maintains for the sake of the structures it is processing. But since we have already treated all such potential interactions, the last line of S4-949 can stand.

Indeed, it might as well, since an explicit version would be no more illuminating. We would have to replace the last line by something of the following form:

```
(SELECT PROC↕                                                    (S4-950)
   [↑+  ↑(+ ↓(1ST ARGS) ↓(2ND ARGS))]
   [↑=  ↑(= ↓(1ST ARGS) ↓(2ND ARGS))]
   ... )
```

which is hardly elucidating. One subtle point can be made: the approach followed in S4-949 de-references the arguments *en masse*; the protocol just suggested de-references them one-by-one. That these both work turns on the fact that 1ST and 2ND and so forth work on rails. In particular, suppose we were normalising (+ 2 3). Then PROCEDURE would designate the primitive addition closure, and ARGS would be bound to the handle '[2 3]. The full ↓ARGS would normalise to [2 3]; similarly, ↓(1ST ARGS) would simplify through ↓'2 to 2, and ↓(2ND ARGS) similarly to 3. Hence the two approaches are equivalent.

The only other main processing function is NORMALISE-RAIL:

```
(DEFINE NORMALISE-RAIL                                           (S4-951)
   (LAMBDA EXPR [RAIL ENV CONT]
      (IF (EMPTY RAIL)
          (CONT (RCONS))
          (NORMALISE (1ST RAIL) ENV
             (LAMBDA EXPR [ELEMENT!]
                (NORMALISE-RAIL (REST RAIL) ENV
                   (LAMBDA EXPR [REST!]
                      (CONT (PREP ELEMENT! REST!)))))))))
```

This is a straight-forward left-to-right tail-recursive (i.e., iterative) normaliser. Worth noting are the use of (RCONS) at the end, rather than '[], so that every normal-form rail is not given the same foot. In addition, it is important that NORMALISE-RAIL is called with the "rest" of the rail, rather than NORMALISE; the difference is that if the general NORMALISE were called, an explicit check for normal-formedness would be executed each time. We do not care so much about the inefficiency here as with the fact that, if some tail *were* determined to be in normal-form, that whole tail would be returned as is. In other words, if X were bound to 1, the rail [X 2 3] would normalise to [1 2 3], as expected, but the first tail of the original rail and of the returned rail would be *actually the same* (with potential side-effect consequences). It would also mean, for example, that NORMALISE-RAIL would not need to check for the empty case, since all empty rails are in normal-form. The adopted strategy, however, is that if *any* of the elements of a rail are not in normal-form, an entirely new rail is returned as the normal-form result.

It follows from S4-961 in conjunction with our use of rails for multiple arguments that we are committed to to a left-to-right order of processing arguments. Often meta-circular processors do not reveal this: if *they* were processed right-to-left, then the processor they implement would process arguments in a right-to-left fashion as well. However this property is not true of our definition, since the processing of the remainder of a rail is wrapped in a continuation (deferred with our standard technique).

This then completes the processor. We also give a definition of the top-level READ-NORMALISE-PRINT. It is assumed that this is initially called with an appropriately initialised standard environment, which is then handed around the loop each time (we discuss initialisation below):

```
(DEFINE READ-NORMALISE-PRINT                                    (S4-962)
   (LAMBDA EXPR [ENV]
      (BLOCK (PROMPT)
             (LET [[NORMAL-FORM (NORMALISE (READ) ENV ID)]]
                (BLOCK (PROMPT)
                       (PRINT NORMAL-FORM)
                       (READ-NORMALISE-PRINT ENV))))))
```

Of interest here is the fact that ID — the identity function — is given as the continuation to NORMALISE, with the printing of the result *outside* the scope of that continuation. Although this does not matter crucially here, we will see in 3-LISP how this affects the interaction between reflective procedures and user interaction. Also we may note the semantic appropriateness of READ and PRINT: they return and expect arguments *designating* structures, respectively, which is just what NORMALISE expects and returns. Thus the two mesh without complicated level-shifting.

In the remainder of the section we will briefly present the utility functions that underwrite the workings of this meta-circular processor, as well as giving defintions for some other utilities of a similar nature that have been used in prior examples. We begin with the identity function, used primarily as a continuation to NORMALISE or REDUCE to "flip" the answer out to the caller of the NORMALISE or REDUCE redex:

```
(DEFINE ID (LAMBDA EXPR [X] X))                                 (S4-963)
```

Four simple vector selectors:

```
(DEFINE 1ST (LAMBDA EXPR [X] (NTH 1 X)))                        (S4-964)
(DEFINE 2ND (LAMBDA EXPR [X] (NTH 2 X)))
(DEFINE 3RD (LAMBDA EXPR [X] (NTH 3 X)))
(DEFINE 4TH (LAMBDA EXPR [X] (NTH 4 X)))
```

Two common tail selectors: the first and the last:

```
(DEFINE REST (LAMBDA EXPR [VEC] (TAIL 1 VEC)))                    (S4-955)
(DEFINE FOOT (LAMBDA EXPR [VEC] (TAIL (LENGTH VEC) VEC)))
```

Two common predicates on vector lengths:

```
(DEFINE EMPTY (LAMBDA EXPR [VEC] (= (LENGTH VEC) 0)))             (S4-956)
(DEFINE UNIT  (LAMBDA EXPR [VEC] (= (LENGTH VEC) 1)))
```

Ten type predicates:

```
(DEFINE ATOM        (LAMBDA EXPR [X] (= (TYPE X) 'ATOM)))         (S4-957)
(DEFINE RAIL        (LAMBDA EXPR [X] (= (TYPE X) 'RAIL)))
(DEFINE PAIR        (LAMBDA EXPR [X] (= (TYPE X) 'PAIR)))
(DEFINE NUMERAL     (LAMBDA EXPR [X] (= (TYPE X) 'NUMERAL)))
(DEFINE HANDLE      (LAMBDA EXPR [X] (= (TYPE X) 'HANDLE)))
(DEFINE BOOLEAN     (LAMBDA EXPR [X] (= (TYPE X) 'BOOLEAN)))

(DEFINE NUMBER      (LAMBDA EXPR [X] (= (TYPE X) 'NUMBER)))
(DEFINE SEQUENCE    (LAMBDA EXPR [X] (= (TYPE X) 'SEQUENCE)))
(DEFINE TRUTH-VALUE (LAMBDA EXPR [X] (= (TYPE X) 'TRUTH-VALUE)))

(DEFINE FUNCTION    (LAMBDA EXPR [X] (= (TYPE X) 'FUNCTION)))
```

A closure is primitive if it is in the following *rail*:

```
(DEFINE PRIMITIVE                                                 (S4-958)
   (LAMBDA EXPR [CLOSURE]
     (MEMBER CLOSURE
           ↑[TYPE = + * - / PCONS SCONS RCONS CAR CDR LENGTH NTH PREP
             TAIL RPLACA RPLACD RPLACN RPLACT LAMBDA EXPR MACRO IMPR
             NAME REFERENT SET READ PRINT TERPRI IF NORMALISE REDUCE])))
```

BINDING designates the binding of a variable in an environment:

```
(DEFINE BINDING                                                   (S4-959)
   (LAMBDA EXPR [VAR ENV]
      (COND [(EMPTY VAR) (ERROR "Unbound variable")]
            [(= VAR (1ST (1ST ENV))) (2ND (1ST ENV))]
            [$T (BINDING VAR (REST ENV))])))
```

Three selector functions on closures:

```
(DEFINE ENV     (LAMBDA EXPR [CLOSURE] ↓(1ST (CDR CLOSURE))))     (S4-960)
(DEFINE PATTERN (LAMBDA EXPR [CLOSURE] ↓(2ND (CDR CLOSURE))))
(DEFINE BODY    (LAMBDA EXPR [CLOSURE] ↓(3RD (CDR CLOSURE))))
```

The type function for distinguishuing procedure types (from S4-248):

```
(DEFINE PROCEDURE-TYPE                                            (S4-961)
   (LAMBDA EXPR [PROCEDURE]
      (SELECT (CAR PROCEDURE)
         [↑EXPR 'EXPR]
```

```
        [↑IMPR 'IMPR]
        [↑MACRO 'MACRO]]))
```

And the XCONS mentioned in section 4.b.iii that facilitates the explicit construction of closures:

```
(DEFINE XCONS                                                    (S4-963)
   (LAMBDA EXPR ARGS
      (PCONS (1ST ARGS) (RCONS . (REST ARGS)))))
```

The BIND procedure used to extend environments upon the expansion of closures. Note that it is MATCH, a subsidiary, that performs the recursive decomposition of non-atomic patterns (MATCH was first introduced in S4-517).

```
(DEFINE BIND                                                     (S4-964)
   (LAMBDA EXPR [PATTERN ARGS ENV]
       ↓(JOIN ↑(MATCH PATTERN ARGS) ↑ENV)))

(DEFINE MATCH                                                    (S4-965)
   (LAMBDA EXPR [PATTERN ARGS]
      (COND [(ATOM PATTERN) [[PATTERN ARGS]]]
            [(HANDLE ARGS) (MATCH PATTERN (MAP NAME ↓ARGS))]
            [(AND (EMPTY PATTERN) (EMPTY ARGS)) (SCONS)]
            [(EMPTY PATTERN) (ERROR "Too many arguments supplied")]
            [(EMPTY ARGS) (ERROR "Too few arguments supplied")]
            [$T ↓(JOIN ↑(MATCH (1ST PATTERN) (1ST ARGS))
                       ↑(MATCH (REST PATTERN) (REST ARGS)))])))
```

As opposed to BIND, REBIND smashes the current binding in an environment, or adds it to the end if there was none there (see section 4.c.vi). The check for normal-formedness of the binding is done just once.

```
(DEFINE REBIND                                                   (S4-966)
   (LAMBDA EXPR [VAR BINDING ENV]
      (IF (NORMAL BINDING)
          (REBIND* VAR BINDING ENV)
          (ERROR "Binding is not in normal form"))))

(DEFINE REBIND*                                                  (S4-967)
   (LAMBDA EXPR [VAR BINDING ENV]
      (COND [(EMPTY ENV) (RPLACT 0 ↑ENV ↑[[VAR BINDING]])]
            [(= VAR (1ST (1ST ENV)))
             (RPLACN 2 ↑(1ST ENV) ↑BINDING)]
            [$T (REBIND* VAR BINDING (REST ENV))])))
```

We include our side-effect version of the fixed-point function:

```
(DEFINE Z                                                        (S4-968)
   (LAMBDA EXPR [FUN]
      (LET* [[TEMP (LAMBDA EXPR ARGS
                     (ERROR "Partially constructed closure reduced"))]
             [CLOSURE ↑(FUN TEMP)]]
```

```
            (BLOCK (RPLACA ↑TEMP (CAR CLOSURE))
                   (RPLACD ↑TEMP (CDR CLOSURE))
                   TEMP))))
```

and the version of DEFINE that uses it (from S4-727):

```
(DEFINE DEFINE                                                    (S4-969)
   (PROTECTING [Z]                          ; Since there is a tendency
      (LAMBDA MACRO [LABEL FORM]            ; to reset Z in examples!
         `(SET ,LABEL
               (,↑Z (LAMBDA EXPR [,LABEL] ,FORM))))))
```

Because we are using the primitive closures (in the meta-circular processor's *own* environment) to mark procedures we will primitively reduce, the following procedure will yield an appropriately initialised standard environment (encoding of E_0). (There is actually an incompleteness here: the *closures* that are the bindings here will have the meta-circular processor's own E_0 as their first argument, rather than this one, but since these are recognised primitively this won't matter. We will cure this inelegance in 3-LISP.)

```
(DEFINE INITIAL-ENVIRONMENT                                       (S4-970)
   (LAMBDA EXPR []
      [['TYPE   ↑TYPE]   ['=      ↑=]      ['+      ↑+]
       ['*      ↑*]      ['/      ↑/]      ['-      ↑-]
       ['PCONS  ↑PCONS]  ['RCONS  ↑RCONS]  ['SCONS  ↑SCONS]
       ['CAR    ↑CAR]    ['CDR    ↑CDR]    ['LENGTH ↑LENGTH]
       ['NTH    ↑NTH]    ['TAIL   ↑TAIL]   ['PREP   ↑PREP]
       ['RPLACA ↑RPLACA] ['RPLACD ↑RPLACD] ['RPLACN ↑RPLACN]
       ['RPLACT ↑RPLACT] ['LAMBDA ↑LAMBDA] ['EXPR   ↑EXPR]
       ['IMPR   ↑IMPR]   ['IF     ↑IF]     ['NAME   ↑NAME]
       ['SET    ↑SET]    ['READ   ↑READ]   ['PRINT  ↑PRINT]
       ['TERPRI ↑TERPRI] ['REDUCE ↑REDUCE] ['NORMALISE ↑NORMALISE]
       ['REFERENT ↑REFERENT]]))
```

Given this definition, the 2-LISP processor could be "started up" by executing

```
(READ-NORMALISE-PRINT (INITIAL-ENVIRONMENT))                      (S4-971)
```

A simple prompter:

```
(DEFINE PROMPT                                                    (S4-972)
   (LAMBDA EXPR [] (BLOCK (TERPRI) (PRINT '>))))
```

MEMBER is defined over both kinds of vector. Note that we don't need to distinguish a special "EQ" version, as we did in 1-LISP:

```
(DEFINE MEMBER                                                    (S4-973)
    (LAMBDA EXPR [ELEMENT VECTOR]
       (COND [(EMPTY VECTOR) $F]
             [(= ELEMENT (1ST VECTOR)) $T]
             [$T (MEMBER ELEMENT (REST VECTOR))])))
```

All elements of three of the structure types are normal-form inherently; atoms are *never* in normal form, and certain rails and pairs can be:

```
(DEFINE NORMAL                                                    (S4-974)
    (LAMBDA EXPR [X]
       (SELECTQ (TYPE X)
          [NUMERAL $T]
          [BOOLEAN $T]
          [HANDLE  $T]
          [ATOM    $F]
          [RAIL (AND . (MAP NORMAL X))]
          [PAIR (AND (MEMBER (CAR PAIR) ↑[EXPR IMPR MACRO])
                     (NORMAL (CDR PAIR)))])))
```

A simple rail copying procedure:

```
(DEFINE COPY                                                      (S4-976)
   (LAMBDA EXPR [RAIL]
      (IF (EMPTY RAIL)
          (RCONS)
          (PREP (1ST RAIL) (COPY (REST RAIL))))))
```

The following two rail conjoiners were first illustrated in section 4.b.vii (see in particular S4-349 and S4-360):

```
(DEFINE JOIN                                                      (S4-976)
   (LAMBDA EXPR [RAIL1 RAIL2]
      (RPLACT (LENGTH RAIL1) RAIL1 RAIL2)))

(DEFINE APPEND
   (LAMBDA EXPR [RAIL1 RAIL2]
      (JOIN (COPY RAIL1) RAIL2)))
```

Finally a variety of useful macros. LET and LET* were explained in section 4.c.i:

```
(DEFINE LET                                                       (S4-977)
    (LAMBDA MACRO [LIST BODY]
       `((LAMBDA EXPR ,(MAP 1ST LIST) ,BODY)
         .,(MAP 2ND LIST))))

(DEFINE LET*                                                      (S4-978)
   (LAMBDA MACRO [LIST BODY]
      (IF (EMPTY LIST)
          BODY
          `((LAMBDA EXPR ,(1ST (1ST LIST))
                ,(LET* (REST LIST) BODY))
            .,(2ND (1ST LIST))))))
```

SELECT and SELECTQ: extensional and intensional case dispatches:

```
(DEFINE SELECTQ                                                   (S4-979)
    (LAMBDA MACRO ARGS
       `(LET [[SELECT-KEY ,(1ST ARGS)]]
```

```
            ,(SELECTQ* (REST ARGS)))))

(DEFINE SELECTQ*                                                    (S4-980)
    (LAMBDA EXPR [CASES]
       (COND [(EMPTY CASES) (RCONS)]
             [(= (1ST (1ST CASES)) '$T)
              (2ND (1ST CASES))]
             [$T `(IF (= SELECT-KEY ,↑(1ST (1ST CASES)))
                      (BLOCK . ,(REST (1ST CASES)))
                      ,(SELECTQ* (REST CASES)))])))
```

BLOCK is the 2-LISP sequencer:

```
(DEFINE BLOCK (LAMBDA MACRO ARGS (BLOCK* ARGS)))                    (S4-981)

(DEFINE BLOCK*                                                      (S4-982)
    (LAMBDA EXPR [ARGS]
       (COND [(EMPTY ARGS) (ERROR "Too few arguments")]
             [(UNIT ARGS) (1ST ARGS)]
             [t `((LAMBDA EXPR [?]
                      ,(BLOCK* (REST ARGS)))
                  ,(1ST ARGS))])))
```

The ever-useful COND. Note that though it is tempting COND* cannot itself use COND:

```
(DEFINE COND (LAMBDA MACRO ARGS (COND* ARGS)))                      (S4-983)

(DEFINE COND*                                                       (S4-984)
    (LAMBDA EXPR [ARGS]
       (IF (EMPTY ARGS) (RCONS)
           `(IF ,(1ST (1ST ARGS))
                ,(2ND (1ST ARGS))
                ,(COND* (REST ARGS))))))
```

Finally, the PROTECTING macro introduced without definition in section 4.c.vi, and used in defining DEFINE above:

```
(DEFINE PROTECTING                                                  (S4-985)
   (LAMBDA MACRO [NAMES BODY]
      `(LET ,(PROTECTING* NAMES) ,BODY)))

(DEFINE PROTECTING*                                                 (S4-986)
   (LAMBDA EXPR [NAMES]
      (IF (EMPTY NAMES)
          (RCONS)
          (PREP `[,(1ST NAMES) ,(1ST NAMES)]
                (PROTECTING* (REST NAMES))))))
```

We have defined all the utilities used in the meta-circular processor (plus a few more); in the rest of this section we will define an additional set that were used in examples, either with or without definitions. These will be considered to be part of the

"kernel" system — we will use them here and in 3-LISP without further introduction.

We begin with two multi-argument boolean connectives that process only as far as necessary, providing the arguments are rails; if not, they allow the entire argument designator to be processed, before returning a result (see S4-918):

```
(DEFINE AND                                                    (S4-987)
   (LAMBDA MACRO ARGS
      (IF (RAIL ARGS) (AND* ARGS) `↓(AND* ↑,ARGS))))

(DEFINE AND*                                                   (S4-988)
    (LAMBDA EXPR [ARGS]
       (IF (EMPTY ARGS)
           '$T
           `(IF ,(1ST ARGS) ,(AND* (REST ARGS)) '$F))))

(DEFINE OR                                                     (S4-989)
   (LAMBDA MACRO ARGS
      (IF (RAIL ARGS) (OR* ARGS) `↓(OR* ↑,ARGS))))

(DEFINE OR*                                                    (S4-990)
    (LAMBDA EXPR [ARGS]
       (IF (EMPTY ARGS)
           '$F
           `(IF ,(1ST ARGS) '$T ,(OR* (REST ARGS))))))
```

We use a MAP that is reminiscent of LISP 1.5's MAPC: it is given successive elements of the sequence (or sequences) on each iteration, and a sequence of results is returned. The FIRSTS and RESTS used by MAP are inefficient but simple:

```
(DEFINE MAP                                                    (S4-991)
    (LAMBDA EXPR ARGS
       (MAP* (1ST ARGS) (REST ARGS))))

(DEFINE MAP*                                                   (S4-992)
    (LAMBDA EXPR [FUN VECTORS]
       (IF (EMPTY VECTORS)
           (FUN)
           (IF (EMPTY (1ST VECTORS))
               (1ST VECTORS)
               (PREP (FUN . (FIRSTS VECTORS))
                     (MAP* FUN (RESTS VECTORS)))))))

(DEFINE FIRSTS                                                 (S4-993)
    (LAMBDA EXPR [VECTORS]
       (IF (EMPTY VECTORS)
           VECTORS
           (PREP (1ST (1ST VECTORS))
                 (FIRSTS (REST VECTORS))))))
```

```
(DEFINE RESTS                                                    (S4-994)
    (LAMBDA EXPR [VECTORS]
       (IF (EMPTY VECTORS)
           VECTORS
           (PREP (REST (1ST VECTORS))
                 (RESTS (REST VECTORS))))))
```

The REDIRECT of S4-367 that did not affect others who held access to the old tail:

```
(DEFINE REDIRECT                                                 (S4-995)
   (LAMBDA EXPR [INDEX RAIL NEW-TAIL]
      (IF (< INDEX 1) (ERROR "REDIRECT called with illegal index")
          (RPLACT (- INDEX 1)
                  RAIL
                  (PREP (NTH INDEX RAIL) NEW-TAIL)))))
```

Finally, a version of PUSH that does not require re-SET-ing in order to be effective, and a corresponding POP. It is assumed that STACK is a rail of items:

```
(DEFINE PUSH                                                     (S4-996)
   (LAMBDA EXPR [ELEMENT STACK]
      (RPLACT 0
              STACK
              (PREP ELEMENT
                    (IF (EMPTY STACK)
                        (RCONS)
                        (PREP (1ST STACK) (REST STACK)))))))

(DEFINE POP
   (LAMBDA EXPR [STACK]
      (IF (EMPTY STACK)
          (ERROR "Stack underflow")
          (BLOCK1 (1ST STACK)
                  (RPLACT 0 STACK (REST STACK))))))
```

4.e. Conclusion

Considering that it was to be a preparatory step in the drive towards reflection, the development of 2-LISP has taken considerable work. Nonetheless we stand in good stead to tackle the intricacies of self reference in the next chapter. We have demonstrated, in addition, that normalisation and reference as independent notions can carry us through the entire range of issues involved in designing a computational formalism.

We said at the very beginning of the chapter that, as well as this primary task, two other goals were sought: the support of higher order functionality and argument objectification within a base language. On these fronts we have succeeded rather well: so long as everything remained extensional, we encountered no problems with the provision of statically scoped higher order functional protocols, and the use of distinct structural categories for redexes and enumerations, either on their own or in interaction. In addition, we were able to provide a full range of meta-structural support: handles, intensional procedures, NAME and REFERENT primitives, macros, and so forth.

However there are a number of ways in which we may have seemed to fail as well. Three stand out as of prime importance. The first was the undeniably awkward interaction that we constantly encountered between non-rail redex CDRS and intensional procedures (both IMPRS and MACROS). Although it is often extremely useful to be able to use designators of a whole argument set rather than one designator per argument, we found it natural, in constructing intensional procedures, to assume that we could decompose the *argument expression*, rather than simply being able to decompose the *argument sequence* in the way that extensional procedures typically do. This is of course not a formal or theoretical difficulty: everything remained perfectly well defined, and a variety of techniques arose naturally to cover the examples we investigated. But it does challenge our assumption that argument objectification and meta-structural concerns are independent and orthogonal. It was of course a criticism that we lodged against 1-LISP that it was forced to use meta-structural machinery for objectifying purposes, and we were indeed able to show that in the standard (extensional) case there was no need for this practice. It is important to realise that the frustration we are currently discussing did not arise from argument objectification on its own, but rather from its *interaction* with bona fide meta-structural manoeuvering.

We seem, in sum, to have ended up with the following conclusion: argument objectification can be adequately treated in an extensional base language without meta-structural machinery. However it makes that base language slightly more complex than it would otherwise be; therefore when meta-structural machinery is introduced (for other reasons), it has to be able to cope with non-syntatically-decomposable argument expressions.

Once put this way, it seems natural enough. Our frustrations may have arisen because we were not able, without modification or even review, to import into our meta-structural practice assumptions as to what was reasonable that were developed in the simpler 1-LISP case. The lesson with regard to this first difficulty, then, is not that we have failed, but that we will need to develop new and more sophisticated meta-structural techniques.

The second "failure", mentioned at the beginnning of the chapter, is more serious. We have come to see how the use of dynamic scoping in 1-LISP, with its natural connection with first order language, facilitates a certain kind of intensional meta-structural practice. It was natural to use static scoping, closures, and the rest, since we wanted a higher-order base language; indeed, our success in this regard implied that we could avoid meta-structural behaviour in many cases where in 1-LISP it would be required. However we encountered an odd consequence of this decision: although it *freed* us from using meta-structural machinery, it also made using meta-structural procedures extremely difficult. The problem was that "once quoted", so to speak, there was no way to "unquote" an expression, *in a way that could recapture its intended significance at the point where it originally occured.* Thus 2-LISP IMPRS looked to be of rather little utility after all. MACROS did not so much solve this problem as provide us with a certain ability to by-pass it, in part because *as part of the definition of macro redex processing* the structure generated by the first phase of macro processing is normalised in the original context. Thus we can see that MACROS solve the problem rather gratuitously.

Though severe, the solution to this problem in 3-LISP was clear: if we could *mention* the processor state, as well as mentioning program structures, then it would be possible completely to overcome this difficulty. Furthermore, as the examples we looked at suggest, and as the discussion in the next chapter will make clear, the situation one achieves with reflection in this regard is not only far more satisfactory than the impoverished 2-LISP

IMPRS, but it is also superior to the 1-LISP situation where the context was available not because it was reified, but because every program fragment, intensional or meta-structural or whatever, was processed in effectively *the same* context. If everything is one, then you won't suffer from too much disconnection (as 2-LISP did), but you have other problems: it is difficult to avoid tripping over your own feet, in a sense (exemplified for example by unwanted variable name collision and so forth). 2-LISP was cleaner than 1-LISP with regard to context, but it was too separated. In 3-LISP we will retain the cleanliness and give back just the right amount of connection.

The third major failing of 2-LISP had to do with our inability to keep the dialect theory independent — with the fact, in other words, that we were forced, for lack of a theory of functional intension, to provide encodings of environments in closures. This decision unleashed a raft of theoretical questions about the relationship between these encodings and the environments they designated, about the range of side-effects of SET, and so forth. As we said in 4.c.ii, 3-LISP will be somewhat better in this regard, because the relationship between environments and environment designators will be faced directly, because 3-LISP is *inherently* theory-relative in very conception, and because all reflective procedures will bind and pass environment designators as a matter of course. But in spite of all of these facts a certain inelegance will remain, arising from this fundamental theoretical lack on our part. The inclusion of an environment designator in a closure is "over-kill", as we suggested in the text: it is a technique that is guaranteed to provide sufficient information to preserve intensional properties, at the expense of preserving far more information than can reasonably be demanded. Nonetheless, it is a limitation we will have to live with in 3-LISP as well. In addition, as we mentioned in section 4.c.vi, it is a failing with certain advantages: in particular, it certainly facilitates redefinitions in a straightforward manner.

Note that the previous (second) problem — that of constructing potent intensional procedures — may in fact reduce to this same lack of a theory of intensionality. If we could bind the parameters in an intensional procedure not to designators of the *expressions* in the original redex, but to designators of their *intension* — if, in other words, we could have an operator (like Montague's "$\uparrow$") that would render the intension of its argument into the extension of the whole — then we might no longer *need* access to the context in which that argument expression originally occurred. This suggestion is supported as well by the

observation, made originally in connection with LAMBDA, that general computational significance is unleashed not on taking the intension, but on reducing/applying that intension subsequently. Under such an approach, furthermore, IMPRS would finally deserve their name: they would be *intensional* procedures, rather than the *hyper-intensional* procedures that they are in the present scheme.

It is worth just a moment's investigation to see how this would go, since this connection with LAMBDA suggests a manner in which we could unify the two issues. Suppose, merely as a temporary mechanism, that IMPR parameters are bound not to the argument expression as it occured in the original redex, but to *closures* of those argument expressions. In addition, suppose that a revised version of NORMALISE, if given a closure of this form (we can suppose they are especially marked) would *reduce* it with no arguments, rather than simply normalising it. We will call this new version NORMALISE*. In other words, given a definition of SET as follows (SET was an example that illustrated our previous difficulty):

```
(DEFINE SET1                                                    (S4-997)
   (LAMBDA IMPR [VAR BINDING]
      (REBIND VAR (NORMALISE* BINDING) <ENV>)))
```

and a use of it as follows:

```
(LET [[X 3]]                                                    (S4-998)
   (BLOCK (SET1 X (+ X 1))
          X))
```

then the formal parameters VAR and BINDING, rather than being bound in the normal way:

```
VAR       ⇒   'X                          ; This is            (S4-999)
BINDING   ⇒   '(+ X 1)                    ; regular 2-LISP.
```

would instead be effectively bound as follows:

```
VAR       ⇒   ↑(LAMBDA EXPR [] X)         ; This is our        (S4-1000)
BINDING   ⇒   ↑(LAMBDA EXPR [] (+ X 1))    ; new proposal.
```

where the assumption is that these would be normalised in the original context. In other words definition S4-997 and S4-998 would together (on this new proposal) be equivalent to the following:

```
(DEFINE SET2                                ; This version of   (S4-1001)
   (LAMBDA EXPR [VAR BINDING]               ; SET is an EXPR.
      (REBIND VAR (REDUCE VAR '[]) <ENV>)))
```

```
(LET [[X 3]]                                                    (S4-1002)
   (BLOCK (SET₂ ↑(LAMBDA EXPR [] X)
                ↑(LAMBDA EXPR [] (+ X 1)))
          X))
```

In other words, the real bindings of VAR and BINDING in the body of SET would be the following:

```
VAR       ⇒   '(<EXPR> [['X '3] ... ] '[] 'X)                  (S4-1003)
BINDING   ⇒   '(<EXPR> [['X '3] ... ] '[] '(X + 1))
```

Then the original call to NORMALISE* in S4-997, which was converted to the equivalent REDUCE in S4-1001, would in fact yield the correct answer '4.

There is a minor difficulty, however: the *first* argument to REBIND was intended to be the simple handle 'X, not a designator of a closure. In other words SET really wanted *hyper-intensional* access to its first argument. To make sense of this proposal, it turns out, we would want to provide an ability to *extract* the variable name from the closure.

We needn't pursue this — the point is clear. Given that we do not have an adequate theory of intensionality, it will prove much simpler, and more general as well, to provide access to the context explicitly, rather than having the dialect itself try to encapsulate that context around the hyper-intensional forms automatically. In S4-1001 we still had no answer to the question of what environment should be given to REBIND as its third argument, which would require yet further machinery. Finally, with an explicit context argument available, code very similar to that in S4-997 and S4-1001 will be easy to write in 3-LISP. We will have, in particular, the following perfectly adequate 3-LISP definition of SET:

```
(DEFINE SET                                  ; This is 3-LISP   (S4-1004)
   (LAMBDA REFLECT [[VAR BINDING] ENV CONT]
      (NORMALISE BINDING ENV
         (LAMBDA SIMPLE [BINDING!]
            (CONT (REBIND VAR BINDING! ENV))))))
```

Apart from the minor extra complexity having to do with the explicitly available continuation, which will prove useful in other cases, the simplicity and the transparency of S4-1004 certainly rival that of S4-997, with the addition that no further complexity about automatically creating pseudo-intensions needs to be added to the underlying dialect. Therefore in the next chapter we will make no further moves to solve the intensionality problem, and will work entirely with reified contexts.

As well as providing us with a rationalised base on which to build a reflective dialect, the development of 2-LISP has had another advantage. In this chapter we have articulated two different theories of 2-LISP: our general meta-theoretical account, and the tail-recursive meta-circular processor of section 4.d.vii. Although the superficial notation of these two formalisms is rather different, it should be clear that the structure of the descriptions formulated in them has been rather similar (although the meta-cricular processor has had to carry only the procedural load, whereas the λ-calculus account has formulated declarative import as well). In the next chapter we will see yet another theoretical encoding of the structure of a dialect: the reflective model. Because our subject matter is only "procedural" reflection, once again only the procedural consequence will be encoded in this causally connected self-referential theory. In a full reflective calculus, which 3-LISP is not, the full theoretical story, including both declarative and procedural consequence, would be embodied in the general reflective model. Such a goal, however, is for another investigation; we turn now to the simpler procedural case.

Chapter 5. Procedural Reflection and 3-LISP

With 2-LISP in place, we turn now to matters of reflection, and to the design of 3-LISP. The presentation of this new dialect will be straightforward: procedural reflection is comparatively simple, given a rationalised base.

Our strategy will be to approach 3-LISP from two opposite directions. First, we will show how the various aspects of 2-LISP that exhibit inchoate reflective behaviour (the meta-circular processor, NORMALISE and REDUCE, and so forth) fail to meet the full requirements of a reflective capability: we will demonstrate, in particular, that they lack the requisite causal connection with the underlying processor. Second, in investigating a variety of candidates for a reflective architecture, we will in contrast look at some suggestions that are *too* connected: proposals that lack sufficient perspective to be coherently controlled The direction we will then head s towards an acceptable middle ground — towards an architecture in which the full state of both structural field and processor are available within the purview of the reflective procedures, but where those reflective procedures have their own independent processor state out of which to work. In addition, we will see how the various theories of LISP that have permeated the analysis so far — embodied in our meta-theoretical characterisations and meta-circular processors — suggest the form that a reflective dialect might take. These preparations will occupy section 5.a.

Though the solution we will converge on will strike the reader of the previous chapters as relatively obvious, it is important to investigate a variety of alternatives for two reasons. On the one hand, it is not enough to show that our particular reflective architecture satisfies our overarching goals; we must also make clear what design choices have been made in the course of its construction. This is particularly important because nothing in our prior analysis uniquely determines the structure of 3-LISP. More specifically, we will characterise three different styles of viable reflective formalism, all compatible with the overarching reflective mandates, but differing in terms of the extent to which each level is detached from that below it. Though we will select just one of these (the intermediate one) for 3-LISP, we will sketch the advantages and limitations of the others, and will show how a more complex formalism could support more than one

simultaneously. In addition, though we will adopt certain "programming conventions" in our use of 3-LISP, we will show how our particular calculus could be used to support a variety of different structuring protocols.

As well as surveying the range of possible architectures, it is equally important to show how the requirements of reflection rule out a variety of plausible candidates. We cannot prove that our range of solutions is unique or complete — indeed, the goals are not of a sort that real *proof* can be imagined — but there are some apparently simpler proposals that might seem, on shallow consideration, as if they would answer the mandates of reflection. Before we take up the definition of 3-LISP we will have shown that these simpler proposals are inherently inadequate.

Ruling out simpler alternatives is particularly important, given that the solution we will adopt will implicate an infinite hierarchy of partially independent processors (an infinite number of environments and continuation structures at any given point in time). The hierarchy is in some ways like that of a typed logic or set theory, although of course each reflective level of 3-LISP is already an omega-order untyped calculus (as was 2-LISP). *Reflective* levels, in others words, are at once stronger and more encompassing than are the order levels of traditional calculi. It may at first blush seem troublesome that, according to the simplest descriptions of 3-LISP, an infinite amount of activity — an infinite number of bindings and procedure calls — happen between any two steps of any program, independent of that program's level. Nonetheless, the fact that 3-LISP is infinite will not prove troublesome: we will be able to show that only a finite amount of information is at any time encoded in these infinite states, so that 3-LISP is after all a finite machine, even though the most convenient virtual account is of an infinite tower of processes. This analysis should dispell any concern as to whether 3-LISP could be efficiently (or even finitely) constructed in an actual physical device. The appendix contains the code of a simple implementation coded in MACLISP, by way of concrete evidence, but we will also in this chapter discuss more generally what is involved in implementing reflective dialects in a way that is complete, and not incurably inefficient. Though it will take some argument, we will be able to demonstrate the following conclusion: *there is no theoretical or practical reason why 3-LISP could not be made to run as efficiently as any other current dialect.*

3-LISP itself will be introduced in section 5.b. Structurally — that is, from the point of view of the structural field and simple functions defined over it — 3-LISP is identical to 2-LISP. As a consequence, 3-LISP is for the most part already defined. Thus, the structural primitives (PCONS, CAR, CDR, TAIL, SCONS, PREP, and all the rest) will have their same definition, both declaratively and procedurally. All of the work we did in defining 2-LISP — including for example our exploration of the relative identity conditions on rails and sequences — will be carried over intact. The only aspects of 2-LISP that will *change* are the 2-LISP IMPRS and MACROS, both of which will be subsumed under the more general notion of a reflective procedure. It will also turn out that, in virtue of the power of the reflective capabilities, some of what is primitive in 2-LISP (LAMBDA, IF, and SET, for instance) will be definable in 3-LISP.

Although its superficial differences are few, 3-LISP merits its status as a distinct dialect because of the rather major shift in the underlying architecture that it embodies. This change is manifested in its implementation: even the simple (and inefficient) implementation presented in the appendix is several times more complex than a comparably efficient implementation of 2-LISP would be. This complexity of implementation, however, should not be read as implying that the dialect is itself complex — rather, it arises out of the dissonance between the abstract structure of the implementing architecture, as compared with the abstract structure of the implemented architecture (between MACLISP's single processor and 3-LISP's infinite number, to be specific). In fact 3-LISP is in many ways a simpler formalism than 2-LISP. At the end of chapter 4, in formulating 2-LISP, we encountered more and more awkward issues and unresolvable conflicts. 3-LISP, in contrast, is in a much happier state: it has natural and rather complete boundaries. Not only will it provide solutions to all of our problems with 2-LISP (as section 5.b.viii makes clear): in addition, none of the issues we will investigate about the new calculus will bring us into conflict with its own new limits. That 3-LISP is in this way *self-contained* is one of its strongest recommendations, providing indirect evidence that our inclusion of reflective capabilities is indeed the correct solution to a range of programming problems.

One fact about 3-LISP should be kept in mind throughout the discussion. In dealing with reflective architectures it is mandatory to maintain a clear understanding of the relationship between *levels of designation* and *levels of reflection.* When a process reflects, as we will see, what was tacit becomes explicit, and what was used becomes mentioned.

We saw a glimpse of this in the 2-LISP IMPRS, where the arguments that were used in the applications were mentioned by the formal parameters in the body of the IMPR procedure itself. When we come to make substantial use of 3-LISP reflective procedures, we will encounter many more examples of such level-crossing protocols. In order to keep this all straight, we will depend heavily on a feature we (not accidentally) built into 2-LISP: the 2-LISP processor is "semantically flat", in the sense that it stays at a fixed semantical level (neither "referencing" nor "de-referencing") in the normal course of events. Because of this, by and large, 3-LISP's *reflective level* (how many steps it has backed off from "reasoning" about the user's world) can be aligned with the *meta-structural level* (how many levels of designation lie between the symbols being used and that user's world). In spite of this correspondence, however, it should be clear that this is not a *tautological* correlation: a reflective version of LISP 1.5 or SCHEME or of any other non-flat calculus *could* be defined, without this property. Indeed, we will allow semantic level-crossing behaviour (using NAME and REFERENT) within a given reflective level, even though we also enforce a semantical level crossing between reflective levels. Our point is that the semantic flatness of the 2-LISP processor will be useful in helping us keep use and mention straight as we cross reflective boundaries, not that the semantic and reflective boundaries are the same thing.

Finally, there are two ways in which our analysis of 3-LISP is incomplete. First, in this chapter we will almost entirely avoid any mathematical analysis, for the simple reason that this author has not yet developed a mathematical approach to reflection that would enable us to characterise 3-LISP in any finite way (other than the objectionable solution of effectively mimicking the implementation in the meta-theory). The subject is taken up in section 5.e, where some suggestions are explored as to what would be involved. This lack of mathematical power is one of the reasons we designed 2-LISP first, for 2-LISP was a formalism in which our mathematics (other than the one issue of environment encodings) was at least partially adequate. In the present chapter we will be forced to rely solely on conceptual argumentation and example.

Second, although we will define a "complete" version of 3-LISP, it will take substantial further research to explore the best ways in which to use the architecture in a raft of different situations. Many suggestions will arise in this chapter that have not yet been fully explored; section 5.d, for example, introduces but does not follow through on half a dozen programming problems where solutions involving reflective abilities seem

indicated. We think of 3-LISP more as a kit or laboratory in which to investigate practical uses of reflection, rather than as an already instantiated system. We will conclude with an explicit discussion of "directions for future research" in chapter 6, but many of the issues requiring additional investigation will emerge in the present chapter.

5.a. The Architecture of Reflection

In the prologue and in chapter 1 we identified a number of properties that any reflective dialect must possess: the ability at any point to step back from the course of a computation to consider what was being done, the power to reach a decision that would influence the future course of that computation, and so forth. We said as well that reflection is inherently theory-relative — that in order to contemplate one's prior state one must have a theory with respect to which that state is described. It has been clear throughout that the meta-circular processors of 1-LISP and 2-LISP are *approximately* self-descriptive theories (albeit of a procedural variety, but since we are in pursuit of procedural reflection that will suffice), but we have implied as well that their obvious encodings lack some crucial properties that a reflective theory would have to have — of which adequate *causal connection* and *sufficient detachment* are two of primary importance. Our analysis in this section of candidate proposals for a reflective architecture, therefore, will focus primarily on their respective merit with respect to these two properties. A great many of the technical problems in reflection, in other words, are best viewed as facets of two general issues: *where you stand* and *what you have access to.*

5.a.i. The Limitations of 2-LISP

We will first show how 2-LISP's processor primitives, and its meta-circular processor, both fail to be reflective. We will assume, for discussion, that the names NORMALISE and REDUCE are bound to the primitive closures designating the actual processor functions, as assumed in chapter 4 (or to definitions in terms of NAME and REFERENT as suggested in S4-852 and S4-853 — it makes no difference), and that MC-NORMALISE and MC-REDUCE are the names of meta-circular versions of these functions, defined approximately as follows (fuller definitions were given in S4-945 and S4-946; the attendant utilities were defined in section 4.d.vii as well):

```
(DEFINE MC-NORMALISE                                                    (S6-1)
   (LAMBDA EXPR [EXP ENV CONT]
      (COND [(NORMAL EXP) (CONT EXP)]
            [(ATOM EXP) (CONT (BINDING EXP ENV))]
            [(RAIL EXP) (MC-NORMALISE-RAIL EXP ENV CONT)]
            [(PAIR EXP) (MC-REDUCE (CAR EXP) (CDR EXP) ENV CONT)])))
```

```
(DEFINE MC-REDUCE                                                    (S6-2)
   (LAMBDA EXPR [PROC ARGS ENV CONT]
      (MC-NORMALISE PROC ENV
           (LAMBDA ... ))))
```

The first task is to make clear how NORMALISE and MC-NORMALISE differ.

There are four facets of a computational process: field, interface, environment, and continuation. NORMALISE and MC-NORMALISE are equivalent with respect to the first two of these, but different with respect to the third and fourth. More specifically, in terms of the field and interface, NORMALISE and MC-NORMALISE are identical not only to each other but also to the underlying processor: they have causal access to these parts of their embedding context *tacitly*, in virtue of their existence as normal programs. This access is illustrated in the following two sessions:

```
> (PRINT 'HELLO) HELLO                                               (S6-3)
> $T
> (NORMALISE '(PRINT 'HELLO)) HELLO
> $T
> (MC-NORMALISE '(PRINT 'HELLO) GLOBAL ID) HELLO
> $T
```

Thus printing causes the same behaviour, whether engendered directly, by a call to NORMALISE, or by a call to MC-NORMALISE (as is reading). Similarly, a request to modify the field given to any of these three processors will have the same result:

```
> (SET X '[INELUCTABLY AMICABLE])                                    (S6-4)
> X
> (XCONS 'RPLACN '1 ↑X ''INEXORABLY)
> '(RPLACN 1 '[INELUCTABLY AMICABLE] 'INEXORABLY)
> (MC-NORMALISE (XCONS 'RPLACN '1 ↑X ''INEXORABLY) GLOBAL ID)
> ''INEXORABLY
> X                                      ; Even though MC-NORMALISE was
> '[INEXORABLY AMICABLE]                 ; used, X has changed.
> (NORMALISE '(RPLACN 1 X 'INEXTRICABLY))
> INEXTRICABLY                           ; Similarly, NORMALISE can also
> X                                      ; be used to change X.
> '[INEXTRICABLY AMICABLE]
> (RPLACN 2 X 'CONFUSED)
> X                                      ; Finally, X can of course be
> X                                      ; changed directly.
> '[INEXTRICABLY CONFUSED]
```

For discussion, we will say that NORMALISE and MC-NORMALISE ***absorb*** the field and interface from their embedding context.

Nothing requires the absorption of field and interface: a meta-circular processor, as suggested informally in chapter 2, is merely one of a variety of possible procedural self-

models. A full implementation of 2-LISP in 2-LISP would more likely explicitly mention at least the field, and possibly the interface as well. For example, we could readily construct a procedure, called IMP-NORMALISE, that required explicit arguments designating all four of these theoretical posits. We would have to decide on some format for designating the structural field and input/output streams in s-expressions (including a decision as to what the appropriate normal-form designators would be, which would depend in turn on what ontological structure we took those entities to have); we would then call IMP-NORMALISE with redexes of the form

```
(IMP-NORMALISE <EXP> <ENV> <CONT> <FIELD> <INPUT/OUTPUT>)          (S5-6)
```

Primitive functions like RPLACA would call the continuation with a different FIELD argument than that with which they were called, and so forth. One could imagine, for example, code for treating RPLACA of roughly the following sort (assuming that the field was represented, as in our meta-theory, as a five-element sequence of functions that mapped structures onto the appropriately related s-expression):

```
(SELECT PROC!                                                       (S5-6)
   ...
   [↑RPLACA [(1ST ARGS)                        ; The result
             ENV                               ; The (unchanged) environment
             [(LAMBDA EXPR [PAIR]              ; The five-part field, with
                (IF (= PAIR (1ST ARGS))        ; the 1st coordinate changed
                    (2ND ARGS)                 ; to encode a new CAR rela-
                    ((NTH 1 FIELD) PAIR)))     ; tionship for this argument.
              (NTH 2 FIELD)                    ; The CDR, FIRST, RFST, and
              (NTH 3 FIELD)                    ; property-list coordinates
              (NTH 4 FIELD)                    ; remain intact.
              (NTH 5 FIELD)]
             INTERFACE]]                       ; The (unchanged) interface
   ... ))
```

Such an implementation would look very much like a straightforward encoding of the mathematical meta-language description of Γ (not Σ, since of course no denotations would be relevant).

With respect to the state of the processor, NORMALISE and MC-NORMALISE differ: MC-NORMALISE requires explicit environment and continuation arguments, whereas NORMALISE also absorbs these aspects of the processor from the tacit context. Furthermore, it is in general impossible to provide particular arguments to MC-NORMALISE so that it exactly mimics NORMALISE. The basic problem is that in 2-LISP there is no way in which one can

obtain (designators of) the actual environments and continuations that are in force during the normal course of a computation. Continuations, in particular, are completely inaccessible; our inelegant practice of including environment designators in closures could be utilised to obtain an environment designator in certain circumstances, but as we will see in a moment even this trick is difficult to parlay into a generally useful protocol.

We will say, in those circumstances in which we *can* obtain causally connected designators of the appropriate facets of a process, that those theoretical entities are ***reified***. Thus the construction of a closure by primitive LAMBDA *reifies* the environment in force at the point of reduction of the LAMBDA (we will see considerably more powerful reification facilities as our investgation proceeds).

It is instructive to show why we cannot construct appropriate reified context arguments for MC-NORMALISE. We start with a very simple case. (The ID function here is the identity function (LAMBDA EXPR [X] X) of S4-953. Note that we write (NORMALISE 'X . .), not (NORMALISE X ...) — an entirely different matter.)

```
(LET [[X 3]] (NORMALISE 'X))        ⇒   '3                              (S5-7)

(LET [[X 3]]
   (MC-NORMALISE 'X [] ID))         ⇒   <ERROR: "X" unbound>
```

The first of the pair works "correctly", so to speak, since the call to the primitively-available NORMALISE makes reference to the same environment (albeit one that is part of the tacit background) that the LET used to bind the X. In the second case the X was *bound* in the same tacit environment, but MC-NORMALISE was given a null environment, for lack of a better alternative, and X was not bound in that environment.

Examples S5-7 can be constrasted with:

```
(NORMALISE '(LET [[X 3]] X))        ⇒   '3                              (S5-8)

(MC-NORMALISE '(LET [[X 3]] X)
              []
              ID)                   ⇒   <ERROR: "LET" unbound>
```

That the first designates the numeral 3 is straightforward: X is bound, as in S5-7, in the tacit encompassing environment, and is looked up in that same environment by NORMALISE. The second, however, doesn't even get started, since LET is not bound in the environment given to MC-NORMALISE. If we had a designator of an environment in which LET was bound to the appropriate macro, we would have:

```
(MC-NORMALISE '(LET [[X 3]] X)                                    (S6-9)
              [['LET ... ] ... ]
              ID)                    ⇒   '3
```

In this case X is bound *in the explicit environment passed around within MC-NORMALISE and MC-REDUCE* — on top, in other words, of the initial environment [['LET ...] ...]. What is shared by the first part of S6-8 and S6-9 is that the binding of X and the lookup of X occur in the same environment, because both expressions are *self-contained*, in a certain sense: they do not depend on the state of the processor external to the call to NORMALISE (or MC-NORMALISE). (In this particular case that lack of dependence is reflected in the fact that they contain no free variables, although there are other forms that dependence can take besides variables.) It is behaviour of this latter sort that motivates us to say that MC-NORMALISE *is equivalent to* NORMALISE.

Note that the continuation function ID did *not* need to be bound in the environment designator passed to MC-NORMALISE in S6-9. This is important: continuations are functions *at the level of the call to the processor*; they are not mentioned in the code that the processor processes.

There are two ways in which the correct environment could be *constructed* in the earlier S6-7. First, we could extract the binding of X from the standard environment and put it in place by hand:

```
(LET [[X 3]]                                                      (S6-10)
   (MC-NORMALISE 'X [['X ↑X]] ID))          ⇒   '3
```

This works because the ENV argument to MC-NORMALISE normalises to the correct sequence [['X '3]]. However this is a gratuitous solution: it could not be generalised except in infinite ways (by, for example, constructing the environment consisting of the present bindings of *all* atoms prior to any call to MC-NORMALISE).

Second, we could tap 2-LISP's inchoate reflective abilities by constructing a dummy closure and ripping the environment designator out of it explicitly:

```
(LET [[X 3]]                                                      (S6-11)
    (LET [[ENV (2ND ↑(LAMBDA EXPR ? ?))]]
       (MC-NORMALISE 'X ENV ID)))          ⇒   '3
```

This is in fact a marginally acceptable solution. It works because closures have to work — we are committed, in other words, by our design of closures, to having the second argument to <EXPR> be a causally connected designator of the environment. However it is

unworkable, as the discussion of GET-ENVIRONMENT at the end of section 4.d.iii made clear. The term (2ND ↑(LAMBDA EXPR ? ?)) will obtain the environment in force only *in the static context in which it is normalised.* Example S6-11 was so simple that this sufficed: in a more complex case, such as to handle the problem of IMPRS, we would like to obtain a designator of an environment in force *at a structurally distal place* (typically, at the point where a redex occured that *calls* the procedure that attempts to obtain that environment).

The situation regarding the continuation structure — the control stack — is analogous: MC-NORMALISE, since it passes a continuation explicitly, forces the continuation structure of the expression being explicitly normalised to be entirely different from that in force when the call to MC-NORMALISE was made; NORMALISE once again uses the same continuation with which it was called. Furthermore, we have in 2-LISP no other behaviour from which we can extract a continuation designator in the way that we just used closures to obtain an environment designator. Since we do not have fancy control procedures to illustrate this point, we will not give specific examples, but the general problem is easy to envisage. Imagine for instance that a CATCH were wrapped around a call to MC-NORMALISE, and the latter procedure encountered a THROW redex in the middle of an expression that it was processing. Clearly the two would not mate in any simple way. The THROW that MC-NORMALISE processed would match only a prior CATCH *that MC-NORMALISE had been explicitly given.*

The contrast between NORMALISE and MC-NORMALISE is made even clearer by looking at examples where the code being processed explicitly calls the processor. Suppose that the atom GLOBAL designates an appropriately initialised environment containing bindings of all the primitive procedures, of ID, of MC-NORMALISE, and so forth (GLOBAL could be built by extending the result of a call to the INITIAL-ENVIRONMENT procedure of S4-970, for example). Then we would have the following:

```
(NORMALISE                                                        (S5-12)
     '(LET [[X 3]] (NORMALISE 'X)))                  ⇒   '3

(NORMALISE
     '(LET [[X 3]] (MC-NORMALISE 'X GLOBAL ID)))     ⇒   <ERROR>
```

In the second line of this example, X was bound by one processor, but looked up by the other, generating an error. Similarly:

```
(MC-NORMALISE                                                    (S6-13)
     '(LET [[X 3]] (NORMALISE 'X))
     GLOBAL
     ID)                                          ⇒   '3

(MC-NORMALISE                                                    (S6-14)
     '(LET [[X 3]] (MC-NORMALISE 'X GLOBAL ID))
     GLOBAL
     ID)                                          ⇒   <ERROR>
```

If the meta-circular MC-NORMALISE correctly implements the language, then it will implement *NORMALISE* to use the same environment it was passing around; but if it contains yet another call to MC-NORMALISE (assuming MC-NORMALISE was defined in GLOBAL) yet a third implementation is invoked, bearing no causal relationship either to the underlying processor, or the the explicit meta-circular MC-NORMALISE running it.

5.a.ii. Some Untenable Proposals

The present condition, then, is this: the meta-circular MC-NORMALISE and MC-REDUCE, although they are adequately equipped with arguments that fully encode the state of the processor, fail to be reflective because those arguments cannot be causally related to the actual state of the processor that runs the code. The primitive processor functions NORMALISE and REDUCE, on the other hand, are fully connected, but they fail in two other ways: they are not designed to take environments and continuations as arguments, and they are so connected to the basic processing that in using them one encounters collisions and awkwardness — they lack not connectedness but detachment. Furthermore, we still have a raft of procedures — THROW, QUIT, RETURN-FROM, and so forth — that would have to be defined primitively, in terms of the implementation, because neither NORMALISE nor MC-NORMALISE provides sufficient power to define them.

At first blush, these facts suggest a rather simple solution. It would seem natural to provide two primitive functions — called, say, GET-STATE and SET-STATE — that, respectively, return and set the state of the current processor. This is a proposal worth examining for two reasons. First, it is simple — if it were sufficient it should be adopted for that reason alone. In addition, it resembles in certain ways various facilties provided in current dialects whereby a user program can obtain access to the *state of the implementation.* In particular, the "spaghetti stack" protocols of INTERLISP provide almost exactly this functionality. There is, however, a crucial difference: we will define GET-STATE and SET-STATE to traffic in full-fledged LISP structures (with declarative and procedural semantics and all the rest), not in implementation-dependent data structures. Thus the results returned by GET-STATE will be standard structural-field elements: no reference will be made to the structure of the machine on which the dialect of LISP is implemented. Our indictment and ultimate rejection of this proposal, therefore, will hold even more strongly for analogous practices in standard dialects.

Oddly enough, however, this cleanliness of remaining within the structural field makes our suggestion initially *more* confusing than the INTERLISP protocols, rather than *less.* The reason has to do with a difficulty in keeping track of the distinction between structures at one level and structures at the other. One admitted virtue of dealing with regular procedures and with stack pointers, in other words, is that you can tell them apart: the

manner in which they are distinguished has little to recommend it, but *that* you can distinguish them turns out to be useful. In the architecture we will ultimately adopt we too will introduce a system of levels, thereby having both structural homogeneity and disciplined behaviour. The present GET-STATE and SET-STATE proposal, however, will fail in part for lack of any such structuring. In sum, the suggestion we are about to explore is rather a mess: though we will do our best to explain it clearly, any intuition on the part of the reader that GET-STATE and SET-STATE are confusing should be recognised as correct.

The idea is this: normalising a GET-STATE redex would return a LISP structure designating the state of the processor at the moment of reduction. Normalising (GET-STATE), in particular, would be expected to return a two-element rail consisting of designators of the environment and continuation in force at the point of reduction, as follows:

```
(GET-STATE)    ⇒    [[['<atom₁> '<binding₁>]                    (S5-18)
                     ['<atom₂> '<binding₂>]
                      ... ]
                     (<EXPR> ... )]
```

The form of S6-18 is dictated by general 3-LISP facts about normal-form designators, the theoretical posits in terms of which we characterise the state of LISP processors, and so forth. No *new* decisions were required, once the basic functionality of GET-STATE was determined.

Similarly, a parallel function SET-STATE would accept a sequence of the same kind of arguments: rather than proceeding with the course of the computation in effect at the time the application in terms of SET-STATE was executed, the processor would be automatically converted to that encoded in the arguments.

In order to see the consequences of this proposal, suppose we execute the following code:

```
(LET [[X 3] [Y 4]]                                              (S5-19)
    (LET [[[ENV CONT] (GET-STATE)]]
      ... ))
```

The presumption is that ENV will be bound to an environment designator containing, as well as the entire surrounding environment, the just-added bindings for X and Y:

```
ENV  ⇒  [['X '3] ['Y '4] ... ]                                    (S5-20)
```

Similarly CONT will be bound to (a normal-form designator of) the continuation in effect when (GET-STATE) was normalised. To our possible surprise, this continuation (remember all of this is forced) will be one that is ready to accept the result of normalising the argument (GET-STATE), in preparation for binding to the formal parameter structure [ENV CONT]! In other words, suppose the fragment in S5-19 were extended as follows:

```
(LET [[X 3] [Y 4]]                                                (S5-21)
   (LET [[[ENV CONT] (GET-STATE)]]
      (BLOCK (TERPRI)
             (PRINT ↑ENV)
             (CONT '[THATS ALL]))))
```

If we were to normalise this, (GET-STATE) would return an ENV and CONT, which would be bound to ENV and CONT, and the first would be printed (the up-arrow is of course necessary because environments are structures). This would proceed as expected:

```
> (LET [[X 3] [Y 4]]                                              (S5-22)
     (LET [[[ENV CONT] (GET-STATE)]]
        (BLOCK (TERPRI)
               (PRINT ↑ENV)
               (CONT '[THATS ALL]))))
  [['X '3] ['Y '4] ... ]
```

The question, however, is what would happen next. CONT, as we have just agreed, is bound to the continuation ready to accept values for binding to ENV and CONT; once this binding is done, the body of the LET will be normalised. In other words the last line of S5-21 would *restart* CONT (for the second time), this time with two atoms, rather than with legitimate processor state designators. The printing would happen again, and the whole process would cycle, but only once; the second time through would engender a type-error, since CONT would be bound to the atom ALL, which, not designating a function, cannot be reduced with arguments:

```
> (LET [[X 3] [Y 4]]                                              (S5-23)
     (LET [[[ENV CONT] (GET-STATE)]]
        (BLOCK (TERPRI)
               (PRINT ENV)
               (CONT '[THATS ALL]))))
  [['X '3] ['Y '4] ... ]
  'THATS
TYPE-ERROR: REDUCE, expecting a function, was called with the atom ALL
```

The problem is that our CONT is not very useful: it is *still too close* to the call to GET-STATE (it reifies but it still lacks detachment). And furthermore, this would always happen — there is nothing idiosyncratic about our particular example. If GET-STATE were *ever* to be useful, calls to it would have to be embedded within code which unpacked its offering, examined the environment and continuation, and so forth. CONT would always be the continuation ready to do this unpacking, and *that* is not a continuation that one typically wants to reason with or about.

A possible but inelegant solution would be first to agree that GET-STATE would always be bound within the scope of a LET of just the sort illustrated in S5-21 above, and then to define a utility function — called, say, STRIP — that would name CONT (in order to obtain access to it as a structure) and strip off just as many levels of embedding as the LET had added, so as to obtain access to the continuation *with which the* LET *was called.* This is typically what is done when defining debugging functions — RETFUN and so forth — from primitives that merely provide access to the state of the implementing stack: one uses such constructs as (STKPOS -2) and so forth. We could lay out the definition of such a function here, but since we will reject GET-STATE presently, we will avoid it (routines that inspect and decompose continuations, however, will be defined in section 5.d below). However we can illustrate how STRIP would be used:

```
> (LET [[X 3] [Y 4]]                                              (S5-24)
     (LET [[[ENV CONT] (GET-STATE)]]
        (BLOCK (TERPRI)
               (PRINT ↑ENV)
               ((STRIP CONT) '[THATS ALL]))))
  [['X '3] ['Y '4] ... ]
> '[THATS ALL]
```

What is a little odd about this, however, is that it would appear to be merely a complex version of the following:

```
> (LET [[X 3] [Y 4]]                                              (S5-25)
     (LET [[[ENV CONT] (GET-STATE)]]
        (BLOCK (TERPRI)
               (PRINT ↑ENV)
               '[THATS ALL])))
```

It seems, in particular, that if in S5-24 we simply wanted to return [THATS ALL] as the result of the LET, then this latter suggestion — if it works — would be a simpler way of doing that.

Indeed this is the case. Though we have not made it explicit, the presumption throughout the foregoing examples was that not only did the call to GET-STATE *return* (designators of) the environment and continuation that were in effect at the moment of the call, but that this very continuation *received* the result of the call as well. In other words CONT was called with the sequence [ENV CONT] — an oddity that begins to betray why GET-STATE will ultimately be discarded. In other words S5-25, as suspected, would return [THATS ALL] as its result:

```
> (LET [[X 3] [Y 4]]                                              (S5-26)
     (LET [[[ENV CONT] (GET-STATE)]]
        (BLOCK (TERPRI)
               (PRINT ↑ENV)
               '[THATS ALL])))
   [['X '3] ['Y '4] ... ]
> '[THATS ALL]
```

Though this is well enough defined, it solves a problem by avoiding the question. It is beginning to appear as if the continuation returned by GET-STATE will never be used.

This is actually not true: we can imagine a more complex STRIP-like procedure that unwound not only the binding part at the beginning of its use, but that threw away even more of the continuation. For example, we might define a RETURN procedure that would hand (the normalisation of) its argument to the closest enclosing call to some procedure (say, to the nearest BLOCK). RETURN would presumably call STRIP, and then examine continuations one by one in the resulting form until one of the proper form were discovered, which would then be called. For example we might be tempted to define RETURN in approximately the following way:

```
(DEFINE RETURN                                                    (S5-27)
   (LAMBDA EXPR [ANSWER]
      (LET [[CONT (STRIP (1ST (GET-STATE)))]]
         ... look through CONT ... )))
```

However it is not clear we can put the call to STRIP there, since all that this arrangement would do (at best) is to strip off the extra continuation structure *that it put on for its own arguments.* Thus we would need something closer to:

```
(DEFINE RETURN                                                    (S5-28)
   (LAMBDA EXPR [ANSWER]
      (LET* [[[CONT ENV] (GET-STATE)))]
            [CONT (STRIP CONT)]]
         ... look through CONT ... )))
```

Even this, however, will probably not be correct, since the continuation structure added by the call to RETURN may need to be by-passed. We need not work through the details, but this sort of manoeuvring to avoid stepping on your own toes is entirely typical of the use of this kind of self-referential facility (a difficulty we will avoid in 3-LISP).

Another use of a GET-STATE continuation would be as part of a result passed out to a caller, thereby retaining access to the continuation within the scope of this procedure. Again, this is very much like the functionality provided by INTERLISP's spaghetti: everything one could do in that system could be done here.

To see how all of this would go, however, we need to explore the relationship between GET-STATE and SET-STATE, on the one hand, and NORMALISE and MC-NORMALISE, on the other. Note that we have not yet used SET-STATE, although we called CONT as an explicit function. (This last fact, too, should suggest that the proposal under discussion is at least odd — it is not quite clear yet whether SET-STATE will ever be necessary.)

We will look at NORMALISE and SET-STATE in turn. First, if we call MC-NORMALISE with the ENV and CONT that we obtained from GET-STATE, it would seem that we could proceed the computation that was in force. Suppose, for example, we executed the following (note again our use of STRIP, without which we would have the same difficulty we experienced earlier about cycling this code):

```
(LET [[X 10] [Y 20]]                                              (S5-29)
   (LET [[[ENV CONT] (GET-STATE)]]
      (BLOCK (MC-NORMALISE 'X ENV (STRIP CONT))
             (PRINT 'DONE))))
```

Two important observations are provided by this example. First, it is not clear that the call to PRINT will ever happen — or if it happens, when that will be — since CONT, which may at the top include the infinite procedure READ-NORMALISE-PRINT, may never terminate. It is not as if the BLOCK and the pending call to PRINT are thrown away, since MC-NORMALISE is a regular procedure — rather, they are likely to remain hanging forever. If indeed CONT includes this call to READ-NORMALISE-PRINT (which for the moment we will presume), then the call to MC-NORMALISE will never return, even though it will apparently come back to top level:

```
> (LET [[X 10] [Y 20]]                                            (S5-30)
     (LET [[[ENV CONT] (GET-STATE)]]
        (BLOCK (MC-NORMALISE 'X ENV (STRIP CONT))
               (PRINT 'DONE))))
```

```
> 10
> (+ 2 3)
> 5
```

The second observation is related to this odd behaviour. This is that all subsequent normalisation will be processed with one level of implementation intermediating: whereas we assume that all code up to the point of the call to MC-NORMALISE was executed by the primitive processor. In other words, all ensuing computation will be effected not directly by the primitive processor, but in virtue of the primitive processor running MC-NORMALISE. This is presumably unfortunate, since nothing in S5-29 suggests that this deferral of subsequent processing was part of our intent.

We may ask whether SET-STATE answers these troubles. According to our original proposal, SET-STATE takes two arguments — an environment and a continuation — and proceeds the primitive processor with those as its states, rather than with the ones that were in effect at the moment the SET-STATE redex was itself normalised. This of course has a minor bug: we would have to specify, in order to be well-formed, an argument with which the continuation should be reduced: continuations have to be given answers. We will assume, therefore, that SET-STATE takes *three* arguments: an environment, a continuation, and an argument for that continuation (other options are possible, such as providing it with an *expression* and an environment, but they make no material difference here). The natural re-casting of S5-29 under this proposal would be this:

```
(LET [[X 10] [Y 20]]                                                    (S5-31)
    (LET [[[ENV CONT] (GET-STATE)]]
        (BLOCK (SET-STATE ENV (STRIP CONT) (MC-NORMALISE 'X ENV ID))
               (PRINT 'DONE))))
```

Our intent here is to look up the value of X in ENV (the only way we have of doing this is by calling MC-NORMALISE as indicated), and then setting the processor as before.

This is different, sure enough, but once again the call to PRINT would be ignored! The reason is not, in this case, because it would be *pending* for ever, but rather because it would simply be thrown away. The presumption is that SET-STATE is a destructive operation — the state in effect when it was called was supposed to be *replaced* by that encoded in its arguments. This, now that we use it in an example, seems unnecessarily extreme. There are other odd aspects to this decision as well: if CONT, or (STRIP CONT) in our case, is a standard continuation, the ENV argument to SET-STATE is immaterial. In fact,

if anything, it played a role in the call to MC-NORMALISE, not in the call to SET-STATE.

One apparent advantage of S5-31 over S5-29, however, is that subsequent processing is effected by the primitive processor, not by one level of indirection through MC-NORMALISE. The behaviour in the following session is similar to what we had before (in S5-30), but without the concomitant serious inefficiency:

```
> (LET [[X 10] [Y 20]]                                              (S5-32)
     (LET [[[ENV CONT] (GET-STATE)]]
        (BLOCK (SET-STATE ENV (STRIP CONT) (MC-NORMALISE 'X ENV ID))
               (PRINT 'DONE))))
> 10
> (+ 2 3)
> 5
```

We could attempt to repair the design of SET-STATE so as to take an expression, environment, and continuation, and to send to that continuation the result of normalising the expression in the environment. This would seem to rationalise the curious structure of S5-32, yielding something of the following sort:

```
> (LET [[X 10] [Y 20]]                                              (S5-33)
     (LET [[[ENV CONT] (GET-STATE)]]
        (BLOCK (SET-STATE 'X ENV (STRIP CONT))
               (PRINT 'DONE))))
> 10
> (+ 2 3)
> 5
```

Though this seems better, there is a striking fact about this example that we cannot ignore: *the call to SET-STATE looks almost exactly like a call to MC-NORMALISE* — it took exactly the same arguments, and approximately the same behaviour resulted. Thus we must ask how this new SET-STATE *differs* from MC-NORMALISE. To this important question there are two answers: subsequent processing was not indirected, and no pending calls to PRINT were saved forever.

We must keep these two points in mind, but deal with them independently, since they do not seem to bear any inherent relationship to each other. It would seem natural to deal with the first in the following way: since we now have a way in which to obtain access to normal-form designators of environments and continuations, we will posit that the primitively named processor functions NORMALISE and REDUCE *take processor states as arguments*, just as MC-NORMALISE and MC-REDUCE did. We will no longer need the meta-circular versions, since their only use in these last pages has been as variants on NORMALISE

and REDUCE that take these extra arguments. Furthermore, if we use *primitive* procedures we will dispense with our concern about indirect processing: NORMALISE and REDUCE *by definition* perform the required processing directly.

The problem with *this* proposal, however, is that by solving one difficulty (that of deferred processing) it raises another, much more serious one. We admitted explicitly that when SET-STATE was called, it *discarded* the environment and continuation that were in force at the point of reduction: what is far from clear is what happens to the environment and continuation in force when our new NORMALISE is called (that they are somehow maintained is the one thing that distinguishes NORMALISE from SET-STATE in our present configuration). Some examples will suggest that our new proposal is in rather serious trouble in this regard.

First, it seems reasonable to expect that, if given a continuation of ID, that NORMALISE should return its result to the caller (we will have much more to say later about the use of ID as a continuation — it will play a very important role):

```
> (LET [[X 10] [Y 20]]                                          (S5-34)
     (LET [[[ENV CONT] (GET-STATE)]]
        (NORMALISE '(+ X Y) ENV ID)))
> '30
```

Secondly, we would still expect to be able to use CONT directly:

```
> (LET [[[ENV CONT] (GET-STATE)]]                               (S5-35)
     ((STRIP CONT) '100))
> 100
```

However it would seem that the following would be equivalent in effect to S5-34:

```
> (LET [[X 10] [Y 20]]                                          (S5-36)
     (LET [[[ENV CONT] (GET-STATE)]]
        (NORMALISE '(+ X Y) ENV (STRIP CONT))))
> '30
```

This is odd: the last form (NORMALISE '(+ X Y) ENV (STRIP CONT)) is *itself* called with (STRIP CONT) as a continuation, as a quick examination of the definition of LET and of the meta-circular processor will show. One wonders what happens to this pending call to that continuation. Once again, in other words, we have a situation similar to that in S5-33 above:

```
> (LET [[X 10] [Y 20]]                                          (S5-37)
     (LET [[[ENV CONT] (GET-STATE)]]
        (BLOCK (NORMALISE '(+ X Y) ENV (STRIP CONT))
```

```
                (PRINT 'HELLO))))
> '30
```

There would seem to be two options: either the call remains pending forever, or else it is discarded. However it cannot be discarded, for two reasons. First, if it *were* discarded, it would be identical to SET-STATE: we have pretty much admitted that the PRINT redex in the following expression will never be encountered:

```
> (LET [[X 10] [Y 20]]                                              (S5-38)
     (LET [[[ENV CONT] (GET-STATE)]]
        (BLOCK (SET-STATE '(+ X Y) ENV (STRIP CONT))
               (PRINT 'HELLO))))
> '30
```

However there is a more serious reason (we don't *have* to keep NORMALISE different from SET-STATE: we could discard the latter name if necessary): *if* NORMALISE redexes were to discard the context they were processed in, S5-34 would not work: the ID would have no one to give the answer to! Therefore the continuation in S5-37 *must* remain pending until the NORMALISE returns. But this will be to wait forever, since (STRIP CONT) contains an embedded non-terminating call to READ-NORMALISE-PRINT.

Not only will this be an infinite wait, but if it remains pending — and this is the killer argument — there is an implication that there are *two different continuations* being maintained by the underlying processor: the one that is handed to NORMALISE explicitly, and the one that was in force at the point of reduction of the NORMALISE redex. This is the first step down a long slippery slope: if there can be two continuations, there can be an arbitrary number. Suppose for example we were to normalise the following expression:

```
(LET [[[ENV1 CONT1] (GET-STATE)]]                                   (S5-39)
   (BLOCK (NORMALISE '(LET [[[ENV2 CONT2] (GET-STATE)]]
                        (BLOCK (NORMALISE '(+ X Y) ENV2 CONT2)
                               (PRINT 'TWO)))
                     ENV1
                     CONT1)
          (PRINT 'ONE)))
```

There is a question as to whether $CONT_2$ would be bound to $CONT_1$, or to some amalgam of $CONT_1$ and the continuation pending to print "ONE". No answer is immediately forthcoming.

It is time to step back for a moment to see what is going on. In the INTERLISP spaghetti protocols, the continuation structures were implementation dependent constructs, and as such there was no tendency simply to *call* them. Rather, the analogue of SET-STATE had to be used in each case. This has a certain clarity, although the extreme difficulty of

stepping cautiously over and around these continuations while manipulating them was a difficulty, as well as their structural inelegance. When we introduced a protocol in which first-class closures were used to encode the continuation structure, we lost any clear sense of what was running what. The spaghetti protocols are *layered*, in other words: two layers, to be specific, and structurally rather distinct. There seems some evidence that the layering is crucial.

It is also noteworthy that we have not once used the environment structures returned by GET-STATE. There is a reason for this: in a statically-scoped dialect there are many different environments around, with a relatively well-defined protocol dictating which are used in what situation. Because of this isolation of one context from another, the use of GET-STATE did not put us into environment difficulties. It is worth just one example to see how this would not be the case in a standard dynamically scoped LISP. In particular, suppose that we were to embed GET-STATE and SET-STATE into 1-LISP, and that we wanted to define a procedure called DEBUG that was to update a counter each time it was called, and was also to normalise its argument in a modified environment (we assume some function MODIFY-ENVIRONMENT has been appropriately defined). We might imagine something of the following sort:

```
(DEFINE DEBUG                                                    (S5-40)
   (LAMBDA IMPR [ARG]
      (LET [[[ENV CONT] (GET-STATE)]]              ; This is 1-LISP
          (BLOCK (SET 'COUNTER (+ 1 COUNTER))
                 (NORMALISE ARG
                            (MODIFY-ENVIRONMENT ENV)
                            (STRIP CONT))))))
```

We assume that COUNTER is a global variable that is initialised before any calls to DEBUG are normalised. The problem, of course, is that the SET might use a variable name that was in ENV, and affect it. For example, we would have:

```
> (LET [[COUNTER 40]]                                            (S5-41)
     (BLOCK (DEBUG)
            (+ COUNTER 10)))
> 51
```

The reason we raise this has to do with relative isolation: in a statically scoped dialect there are *different* environments in the two redexes S5-40 and S5-41; thus the unwanted collisions are naturally avoided. What is curious about all of our explorations of various continuations in these last pages is that roughly the same sorts of collisions seem to be

troubling us. It is natural to wonder, therefore, whether some analogous solution might be found: a protocol for continuations that bore the same relationship to our current protocols as static scoping bears to dynamic. Obviously it cannot be an isomorphic solution — it is nonsensical to suggest that each reduction that involved the expansion of a closure would use a different continuation: continuations are exactly what tie such redexes together. However it is less clear whether some solution with similar abstract structure might not be found.

5.a.iii. Reflective Code in the Processor

What is *good* about GET-STATE is the fact that it provides access to well-formed normal-form designators of the processor state: a minimal requirement on a reflective facility. What is *bad* about it, however, has to do with the code that is given those designators. In other words we have succeeded in providing a view of the processor, but we have not provided an adequate place to stand in order to do the looking. The troubles in the foregoing examples arose not so much from the results returned by calls to GET-STATE, in other words, but rather in the integration of code using GET-STATE into the processing of regular base-level code. For example, GET-STATE both reified *and* absorbed the continuation from the tacit context. It is all very well to reify it — that has been our goal — but reification should be an *alternative* to absorption, not an *addition.*

In order to see why this is a problem, and from there to identify a better solution, we will look briefly at the revised meta-circular processor that would be required for a dialect in which GET-STATE and our new three-argument NORMALISE were defined. We informally assumed, in the discussions above, that the meta-circular MC-NORMALISE and MC-REDUCE of S5-1 and S5-2 would suffice, even if GET-STATE was defined, but of course the addition of GET-STATE should be manifested in an altered meta-circular interpreter. Furthermore, the second change, whereby NORMALISE and REDUCE were extended to accept three arguments, obviously requires changes to the meta-circular processor as well. The definitions of MC-REDUCE and MC-NORMALISE remain unchanged (providing we assume that SET-STATE and GET-STATE are EXPRS; since the latter takes no arguments, this is as good a choice as any); the differences are manifested in a new definition of REDUCE-EXPR (modifications are underlined):

```
(DEFINE MC-REDUCE-EXPR                                              (S5-46)
   (LAMBDA EXPR [PROC! ARGS ENV CONT]
      (SELECT PROC!
         [↑GET-STATE (CONT ↑[ENV CONT])]
         [↑REFERENT (MC-NORMALISE ↓(1ST ARGS) ENV CONT)]
         [↑SET-STATE (MC-NORMALISE ↓(1ST ARGS) (2ND ARGS) (3RD ARGS))]
         [↑NORMALISE (MC-NORMALISE ↓(1ST ARGS) (2ND ARGS) (3RD ARGS))]
         [↑REDUCE (MC-REDUCE ↓(1ST ARGS) ↓(2ND ARGS)
                             (3RD ARGS) (4TH ARGS))]
         [$T (CONT ↑(↓PROC! . ↓ARGS))])))
```

The definition makes plain the fact alluded to earlier: calls to GET-STATE return as part of their result the same continuation that that result is sent to. As we said above, the difficulties we had arose not over the results themselves, but over the integration of the supposedly reflective code into the main program body. This should make us suspect that CONT is not the ideal continuation to send ↑[ENV CONT] to.

There are some other things to notice about S5-45. First, the level-shifting embodied in all of these protocols is made clear: GET-STATE provides to a continuation (at some level) designators of the environment and continuation *of that same level.* I.e. CONT is called with ↑CONT as an argument. Similarly, SET-STATE de-references its first arguments, but not its second two. Finally, the code for NORMALISE and REDUCE is simply wrong: it is identical to that for SET-STATE. The problem with multiple continuations is made clear in this code: NORMALISE was supposed to save CONT, but it is not clear how this is to be done.

There are limits to pursuing malformed proposals. The crucial insight, to which all of these considerations lead us, is this:

> *Reflective code should be run at the same level as the meta-circular processor; it should not be processed by the meta-circular processor.*

This realisation in one move solves a number of problems: it deals straight away with the ambiguity engendered by the tension between NORMALISE and MC-NORMALISE in the examples in S5-29, S5-31, and S5-34, above, where in one case subsequent code was indirectly processed through the meta-circular processor, whereas in the other it was processed directly. It will also solve all of the problems of integration, as well as the inelegance of the level-shifting involved in such expressions as ↑[ENV CONT] in the code just presented. It is an insight with consequence, however, so we will look at it rather carefully.

The first way to understand it is to take a particular example, rather than attempting to solve the general case. Suppose in particular that we look again at our suggested procedure called DEBUG that was supposed to normalise its arguments in some variety of modified environment. We assume that DEBUG with a single argument should engender the normalisation of something like the following code:

```
(BLOCK (SET COUNTER (+ 1 COUNTER))                              (S5-46)
       (NORMALISE ARG (MODIFY-ENVIRONMENT ENV) CONT))
```

where ARG is assumed to be bound to (a designator of) the arguments provided in a particular application, and ENV and CONT are bound "appropriately" — what this comes to we will see in a moment.

Suppose we construct a special-purpose meta-circular processor to handle this case. I.e., we will not *define* DEBUG, but will instead make it primitive in the processor. This time we will modify MC-REDUCE. (Once again the new code is underlined; in addition, we have to use (1ST ARGS) in place of ARG.)

```
(DEFINE MC-REDUCE                                                    (S5-47)
  (LAMBDA EXPR [PROC ARGS ENV CONT]
    (MC-NORMALISE PROC ENV
      (LAMBDA EXPR [PROC!]
        (IF (EQUAL PROC! ↑DEBUG)
            (BLOCK (SET COUNTER (+ 1 COUNTER))
                   (MC-NORMALISE (1ST ARGS) (MODIFY-ENV ENV) CONT))
            (SELECTQ (PROCEDURE-TYPE PROC!)
              [IMPR (IF (PRIMITIVE PROC!)
                        (MC-REDUCE-IMPR PROC! ↑ARGS ENV CONT)
                        (EXPAND-CLOSURE PROC! ↑ARGS CONT))]
              [EXPR (MC-NORMALISE ARGS ENV
                      (LAMBDA EXPR [ARGS!]
                        (IF (PRIMITIVE PROC!)
                            (MC-REDUCE-EXPR PROC! ARGS! ENV CONT)
                            (EXPAND-CLOSURE PROC! ARGS! CONT))))]
              [MACRO (EXPAND-CLOSURE PROC! ↑ARGS
                        (LAMBDA EXPR [RESULT]
                          (MC-NORMALISE RESULT ENV CONT)))])))))
```

The striking fact, of course, is that ENV and CONT are bound *to their natural bindings within REDUCE;* with just the correct resulting behaviour. Suppose, for example, we look at the following code:

```
(LET [[X 3] [Y 4]]                                                   (S5-48)
    (+ X (DEBUG Y)))
```

and suppose in addition that MODIFY-ENVIRONMENT takes an environment and changes the bindings of all atoms bound to numerals, in such a way that they end up bound to double what they were bound to originally. We would thus get (assuming, of course, that the processor running the following code is the one described by the MC-REDUCE of S5-47):

```
> (LET [[X 3] [Y 4]]                                                 (S5-49)
    (+ X (DEBUG Y)))
> 11
```

This works because the expression (DEBUG Y) is normalised in the course of the computation just as any expression would be: since it is a pair (a redex), it is reduced, causing the

normalisation of the CAR, which yields PROC! (in REDUCE) bound to a designator of the closure of DEBUG. This fact is noticed by the conditional in MC-REDUCE, which then normalises the first argument (Y, in this case) in the modified environment, with the continuation given to the normalisation of (DEBUG Y). This is a continuation that expects normalised arguments for +, since that function is an EXPR; thus 8 is returned to that continuation, and the addition proceeds, yielding 11 as a final answer.

The "place to stand" that we were looking for, in other words, is provided in this example by inserting the code within the meta-circular processor. This is perhaps to be expected, for the meta-circular processor has exactly the properties we have been requiring for reflection: it has its own environments and continuations, but its arguments *designate* the environments and continuations of the code running one level below it.

The open question, however, is how to provide a general solution — our treatment of DEBUG was highly particularised, requiring essentially a private dialect. A first suggestion as an answer is to modify DEBUG as follows: rather than having it primitively recognised by REDUCE, we will posit that it will be categorised as a special type — a REFLECTIVE procedure. Then we will assume that associated with each reflective procedure (of which DEBUG is now just one example) there is another procedure (say, DEBUG* in this case) which is called with the arguments and with the environment and continuation in current force in REDUCE. Finally, we assume that some procedure CORRESPONDING-FUN will embody the mapping between these two (from DEBUG to DEBUG* in our case). The revised definition of REDUCE would be the following:

```
(DEFINE REDUCE                                                    (S5-50)
  (LAMBDA EXPR [PROC ARGS ENV CONT]
    (MC-NORMALISE PROC ENV
      (LAMBDA EXPR [PROC!]
        (SELECTQ (PROCEDURE-TYPE PROC!)
           [REFLECT ((CORRESPONDING-FUN PROC!) ARGS ENV CONT)]
           [IMPR (IF (PRIMITIVE PROC!)
                     (MC-REDUCE-IMPR PROC! ↑ARGS ENV CONT)
                     (EXPAND-CLOSURE PROC! ↑ARGS CONT))]
           [EXPR (MC-NORMALISE ARGS ENV
                    (LAMBDA EXPR [ARGS!]
                       (IF (PRIMITIVE PROC!)
                           (MC-REDUCE-EXPR PROC! ARGS! ENV CONT)
                           (EXPAND CLOSURE PROC! ARGS! CONT))))]
           [MACRO (EXPAND-CLOSURE PROC! ↑ARGS
                     (LAMBDA EXPR [RESULT]
                        (MC-NORMALISE RESULT ENV CONT)))])))))
```

In addition, we have the following definition of DEBUG*:

```
(DEFINE DEBUG*                                                    (S5-61)
   (LAMBDA EXPR [ARGS ENV CONT]
      (BLOCK (SET COUNTER (+ 1 COUNTER))
             (NORMALISE ARG (MODIFY-ENVIRONMENT ENV) CONT))))
```

We are close to a final solution; the present proposal, however, can be simplified substantially. Note that under this new plan the function DEBUG is never defined; we merely *used* expressions of the form (DEBUG <arg>). In addition, DEBUG* is never called explicitly, except in the sixth line of the REDUCE just given. Furthermore, we have not yet indicated how we have indicated that DEBUG is a reflective procedure, nor have we defined CORRESPONDING-FUN. All of these problems can be solved in one move if we adopt the following convention: procedures of type REFLECT (i.e., designated by such expressions as (LAMBDA REFLECT ...) will be recognised by PROCEDURE-TYPE as reflective. When they are *used*, they will be called with a standard set of arguments. Their *definitions*, however, will be, like the definition of DEBUG*, designed to accept three arguments: a *designator* of the arguments provided in an application, an environment, and a continuation. They will be *processed* much as in the case of DEBUG* just given. Thus, for example, under this new plan there would be no function DEBUG*; instead, we would define DEBUG approximately as follows (note the use of pattern decomposition to extract the first argument):

```
(DEFINE DEBUG                                                     (S5-62)
   (LAMBDA REFLECT [[ARG] ENV CONT]
      (BLOCK (SET COUNTER (+ 1 COUNTER))
             (NORMALISE ARG (MODIFY-ENVIRONMENT ENV) CONT))))
```

Exactly how reflective procedures are treated by REDUCE will be explained in detail in section 5.c, but the general flavour is predicted by the foregoing examples.

The final, and perhaps the most important, comment to be made about the definition of REDUCE just given is that it can no longer be fairly called a "meta-circular" processor, in the sense that we were using that term in previous chapters. The problem is that whereas previous versions were merely models of the main processor; runnable but in no way part of the regular processing of expressions, this new definition has a very different status. It would seem as if it would always have to be run, since, when a reflective procedure is invoked, it will actually have to be passed the environments and continuations that have been built up over the course of the computation. In addition, there is no indication of how reflective procedures are in fact treated, since whereas all expressions

treated by the meta-circular processor were previously mentioned (both IMPRS and EXPRS), reflective procedures are in this new version *used* in the meta-circular processor.

Furthermore, reflection should of course be able to recurse. Thus, not only do we seem to have mandated one level of indirected processing, we may in fact have mandated an infinite number of levels.

In spite of these concerns, however (all of which can be taken care of), the suggestion as laid out is essentially the one we will adopt. It has all of the required properties: the full state of the processor is available for inspection and modification, it is fully general, and a natural context is provided for reflective code to run.

There is one final footnote to this long introduction. GET-STATE and SET-STATE have of course disappeared in favour of reflective procedures, but it is trivial to define them as 3-LISP routines, as follows:

```
(DEFINE GET-STATE GLOBAL                                          (S5-63)
   (LAMBDA REFLECT [[] ENV CONT] (CONT ↑[ENV CONT])))

(DEFINE SET-STATE GLOBAL                                          (S5-64)
   (LAMBDA EXPR [ARG ENV CONT]
      ((LAMBDA REFLECT ? (NORMALISE ARG ENV CONT)))))
```

It is readily apparent how GET-STATE returns designators of the environment and continuation to the continuation itself, and how SET-STATE (an EXPR) reflects, ignoring the current context, and proceeding in virtue of the context passed in as an argument.

Finally, the NORMALISE mandated by our new definition has all the properties we wanted: it takes three arguments, *and it maintains continuations* (potentially an infinite number of them, because there are an infinite number of reflective levels). All of this will be made clear in section 5.c.

5.a.iv. Four Grades of Reflective Involvement

We have rather debugged ourselves into an acceptable design, in part by modifying and reacting to the limits of previous suggestions. Before turning to the full development of 3-LISP in accordance with these insights, we must pause to review our progress. What we would like is an *abstract* characterisation of the various proposals that have been rejected, and of the apparently acceptable suggestion we will shortly pursue.

First, the discussion of the meta-circular 2-LISP processor showed us that, *to the extent that the reflective programs pass designators of processor state*, those designators must be causally linked to the actual state they designate. There is no advantage in passing environment and continuation arguments explicitly to MC-NORMALISE if they are not in fact designators of the environment and continuation that were in force. The first principle of reflection, then, is that one must retain adequate causal access.

Second, we saw that there were two different ways in which a processor could have adequate access to a non-reflected process (or aspects of it). In other words there is *more than one way* to have causal access: either by simply being "within" the same context (or field or whatever), or by dealing with structures that *name* that process (or aspects of it). We will call these two kinds of access ***direct*** and ***indirect***, respectively. Thus whereas MC-NORMALISE dealt with designators of environment and continuation, and had no causal access at all; NORMALISE, on the other hand, had *direct* causal access to the environment, continuation, and field. (Too much direct causal access, as we saw, is no better than none.)

It emerged in the discussion of GET-STATE that there is no virtue in having *both* direct and indirect access: only confusion resulted from that suggestion. The problem in that circumstance, in other words, was not only that there was insufficient room for reflective manoeuvring, but also that it was unclear whether the tacit encompassing context, or the one returned as the result of a GET-STATE redex, was to be used in any given situation.

These considerations lead to the following suggestion: reflective procedures should run in the following manner: all of those aspects of the non-reflected procedures that are their subject matter should *either* be given in terms of causally connected indirect (reified) designators, or they should be directly shared with those procedures. We have, approximately, four components to a simple computational process: environment, continuation, field, and input/output interface; thus this mandate would seem to suggest as many as sixteen possible designs, in which each of these four compoments were provided either directly or indirectly to the reflective procedures. Thus the most general theoretical approach would be to define sixteen types of reflective procedure (*procedure* rather than *architecture* because, as we will discuss shortly, a given architecture could support more than one reflective procedure type). However from these sixteen we will select four that

seem the best motivated.

First, there is not much argument — nor much sense — to provide the environment as a direct property (except with regard to publically defined procedures — see section 5.b.iv). Additionally, since the continuation is with the environment part of the state of the *processor*, it is natural to consider them together (i.e., to think of the computational process as consisting of processor and field, and the processor in turn as consisting of environment and continuation, as we have suggested all along). There is no *necessity*, however, to do this: it would be possible to define a dialect that reified the environment but absorbed the continuation. This, after all, is very much like the inchoate use of "environment pointers" in standard LISPs.

What does seem suggested, however, is that it is *more* reasonable to reify the continuation than the whole field, and also that it would be extremely unlikely to want to reify the continuation and *not* the environment. This suggests that we can order the environment, continuation, and field, in terms of candidacy for reification. The interface we will simply for the present ignore, mostly because we have dealt so little with input/output behaviour; its reification (perhaps in the form of streams and so forth) is both straightforward and perhaps less engendering of confusion than any of the other three; in addition, current practice comes closest to reifying it. This leads us, therefore, to the following four types of reflective dialect (we include type 0 for completeness, although it doesn't quite count since it reifies nothing):

(S5-55)

	Environment	Continuation	Structural Field
Type 0 reflection	absorbed	absorbed	absorbed
Type 1 reflection	reified	absorbed	absorbed
Type 2 reflection	reified	reified	absorbed
Type 3 reflection	reified	reified	reified

Type 0 reflection is what the 2-LISP IMPR facility provided: *mention* of procedural fragments, but entirely within the standard processor regimen. NORMALISE and REDUCE (the single argument versions) were similarly of this type 0 variety. As we discovered there, this is an inadequate scheme, and must be rejected: type 0 reflection, in other words, is not really reflection at all.

Type 1 reflection, as mentioned earlier, is not unlike the use of pointers into environments in standard LISPs; we, however, will reject it as well, since it provides no ability to describe continuation (control) aspects of processing, and there is no reason to reject this half of the processor. On the other hand the IMPR problems of 2-LISP would be solved rather nicely by this sort of reflective ability: one imagines an IMPR type of LAMBDA term, binding parameters to argument structure and environment in force at the point of the IMPR redex, but operating within the same continuation structure. Thus for example we might have the following definition of SET (see example S4-806):

```
(DEFINE SET                                                    (S5-56)
   (LAMBDA IMPR [[VAR EXP] ENV]
      (REBIND VAR (NORMALISE EXP ENV ID))))
```

Even more elegant, however, would be to have a version of NORMALISE to use in such circumstances which required only two arguments:

```
(DEFINE SET                                                    (S5-57)
   (LAMBDA IMPR [[VAR EXP] ENV]
      (REBIND VAR (NORMALISE EXP ENV))))
```

We will not adopt this practice in 3-LISP, in order to leave that dialect simple, but two things should be said. First, the strategy we will adopt — reifying environment *and* continuation — is more powerful, so that the behaviour engendered by such IMPRS can always be defined in 3-LISP. It turns out not to be possible to translate syntactically between IMPRS of this sort and more general reflective procedures, for reasons that have to do with subtleties arising from the interaction of continuations. For this reason a practical 3-LISP system might well want to include this sort of type 1 reflection, as well as the more powerful kind we will explore in this chapter.

Finally, there is a choice between type 2 and type 3. The difference would be that reflective procedures, under a type 3 scheme, would receive as an argument a designator of a field, much as the meta-theoretic characterisation of 2-LISP received as argument a field designator. Procedures that accessed components of those fields would have to be provided (again as in the meta-theoretic characterisation): thus we might expect such constructs as (NTH 1 X F) and (LENGTH L1 FIELD-7) and so forth.

We will reject this extra complexity, and proceed instead with developing a type 2 dialect. The reason is merely one of simplicity: there is nothing incoherent about the type 3 proposal, and from a purely mathematical point of view it is perhaps the most elegant.

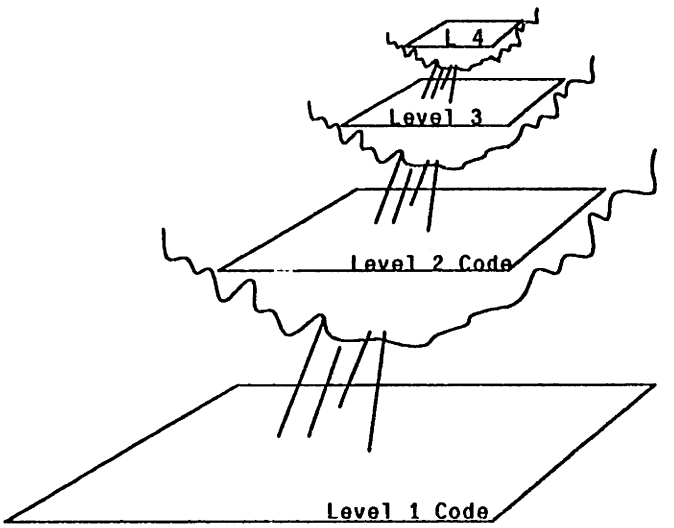
However two facts argue against its adoption. First, it is not clear that there would in a pratical system be much difference between a type 2 and a type 3 dialect, except that the latter would be more complex, because all *access* to the structural field is of two types: either it is direct, in which case it always involves the creation of *new* structure (PCONS and so forth), or else it is *through names*, which are environment relative. By reifying the environment we have reified *access* to the field; thus we are not liable to trip over another level's use of the field unwittingly, by allowing it to remain as a whole undifferentiated among levels. Furthermore, one of the only arguments *for* reifying the field is to save state (so as to be able subsequently to back up a computation); on the other hand, given that a) the field is infinite, and b) we have reified access, we can make a copy of the entire accessible fragment of the field with our current proposal.

Second, in the model of computation in which the structural field was introduced (in chapter 1), it was not presented as *particular* to the processor state, but more as the *world* over which the programs were embedded. It was this fact that led us to characterise all programs as meta-structural; terms in them designate structural field elements. Thus to reify the entire field is not so much to make one process *reflect* as to *implement* an entire computational process in another. This, it would seem, is perhaps *too* much separation.

These are not hard and fast decisions: it is important to recognise the potential viability of both type 1 and type 3 architectures. Nonetheless we will adopt the type 2 proposal in our own design.

A very rough idea of the 3-LISP levels of processor is given in the following diagram. The intent of this picture is to show how each level is processed by an active processor that interacts with it (locally and serially, as usual), but how each processor is in turn composed of a structural field fragment in turn processed by a processor interacting with it.

(S5-58)



Nothing like the detail required to formulate 3-LISP can be conveyed in a simple diagram, of course, but one facet of 3-LISP is indicated here that is crucial to understand. Each processor runs always: there is not a single locus of agency that moves around between levels (even though this is how the *implementation* works, as we will see in section 5.c). Thus it is reasonable to ask at what level a given procedure is run, but it is not reasonable to ask at what level the 3-LISP processor is running.

5.b. An Introduction to 3-LISP

There are two primitive categories of procedure in 3-LISP: ***simple*** and ***reflective***. *Simple* procedures are entirely like 2-LISP EXPRS: they are declaratively extensional, and procedurally, they "normalise their arguments" on reduction, in left to right order. *Reflective* procedures, on the other hand, are more powerful than either 2-LISP IMPRS or 2-LISP MACROS; thus we dispense with both of the latter categories. (It is because reflective procedures are also extensional that we have replaced the name EXPR with SIMPLE.) Procedurally, then, we will deal with two classes of closure. Declaratively we adopt the same semantical domain as in 2-LISP, with its five major categories (structures, truth-values, numbers, sequences, and functions). In addition, the 3-LISP structural field is identical to that of the simpler dialect.

There are twenty-nine primitive 3-LISP procedures, listed in the following table. Those that differ substantially from their 2-LISP counterparts, or that are new in this dialect, are underlined. The table is divided into two parts: the two in the lower half are not strictly primitive, in the same sense that the others are (it is not necessary to have primitively recognised NORMALISE and REDUCE closures bound in the initial environment, for example), but in spite of this their intensional structure is fundamentally integrated into the way that the dialect is defined.

The 3-LISP Primitive Procedures (S6-63)

Arithmetic:	+, -, *, /	—	as usual
Typing:	TYPE	—	defined over 6 syntactic and 4 semantic types
Identity:	=	—	s-expressions, truth-values, sequences, numbers
Structural:	PCONS, CAR, CDR	—	to construct and examine pairs
	LENGTH, NTH, TAIL	—	to examine rails and sequences
	RCONS, SCONS, PREP	—	to construct " " "
Modifiers:	RPLACA, RPLACD	—	to modify pairs
	RPLACN, RPLACT	—	to modify rails
I/O:	READ, PRINT, TERPRI	—	as usual
Control:	IF	—	an extensional if-then-else conditional
Functions:	SIMPLE, REFLECT	—	two primitive kinds of procedure
Semantic:	NAME, REFERENT	—	to mediate between sign and significant

Processor: NORMALISE, REDUCE — primary functions in the reflective processor

Though this complement of primitives is similar to the 2-LISP set, there are a variety of important differences (the section in which the difference is explored is indicated in brackets at the end of each entry):

1. *All 3-LISP primitives are simple.* There are, in other words, no primitive reflectives — no primitives that deal with their arguments intensionally. This not only makes for a rather elegant base; it also simplifies the structure of the reflective processor [5.c]. There are two reflective procedures we might expect to be primitive: LAMBDA and IF. LAMBDA, however, can be defined as a user procedure [5.b.vi], and IF can also be defined, in terms of a simple, applicative order, extensional conditional that we *do* provide primitively, called EF (for *Extensional IF*, since IF can be viewed as the name of an *Intensional IF*) [5.b.ii].
2. There are no naming primitives: SET can be defined, as well as LAMBDA [5.b.iv].
3. The label SIMPLE replaces the label EXPR in all situations in which the latter occurred in 2-LISP. Thus, simple procedures have normal form designators that are redexes formed in terms of the primitive SIMPLE redex; the type argument to a LAMBDA form will in the standard (non-reflective) case be SIMPLE; we will define a PROCEDURE-TYPE predicate to map simple closures onto the atom SIMPLE, and so forth. Thus we would for example have: [5.b.iii]

```
+                                  ⇒   (<SIMPLE> ... )              (S5-64)
((LAMBDA SIMPLE [X] (+ X 1)) 4)    ⇒   5
(PROCEDURE-TYPE ↑TYPE)             ⇒   'SIMPLE
```

4. The four replacing operators (RPLACA, RPLACD, RPLACT, and RPLACN) and the two printing functions (PRINT and TERPRI) can be defined to return no result, although their side-effect behaviour is exactly as in 2-LISP. Contexts, such as in all but the argument position to BLOCK, can be defined to accept such constructs (BLOCK is not primitive). [5.d.i]
5. REFERENT takes an extra (environment) argument. [5.b.vii]
6. NORMALISE and REDUCE, though causally connected to the primitive processor, take three and four arguments, respectively, like the 2-LISP meta-circular processor's versions of these procedures, rather than one and two (like the primitive procedures). [5.c.]
7. ***Reflective redexes*** — redexes whose CARs normalise to ***reflective closures***, which in turn are closures whose CAR is the primitive <REFLECT> closure — are of course processed in an entirely new way, that has no analogue or precedent in 2-LISP. This last difference is the fundamental way in which this dialect is radically distinct from 2-LISP.

From a pedagogical point of view it is a little difficult to introduce 3-LISP, since it is difficult to obtain a deep understanding of reflective procedures without a prior understanding of the reflective processor. On the other hand without *some* understanding of what reflective procedures are, the reflective processor will in its own way make little sense. In the remainder of this section, therefore, we will rather briefly survey those aspects of 3-LISP that are different from 2-LISP, before turning in the next section to an investigation of the processor.

5.b.i. Reflective Procedures and Reflective Levels

Rather than beginning with the new primitives, we will start with the 3-LISP treatment of non-primitive reflective procedures. As was suggested in section 5.a, the body of reflective procedures is intended to be run "one level above" that of the code in which redexes formed in terms of them are found. For example, we illustrated a trivial reflective procedure called DEBUG; if a call to DEBUG is found in code at some level K, then the body of the procedure associated with the name DEBUG is run within the dynamic scope of the reflective processor that runs code at level K: the body, in other words, is run at level K+1.

We assume, in other words, a hierarchy of reflective levels. For convenience alone we number these from 1 to ∞, but there is nothing substantive in the *absolute* values of these numbers: the user, by calling READ-NORMALISE-PRINT explicitly, can create levels with negative indexes. Furthermore, even their ordering is as much a convention as a fact of the 3-LISP architecture; by binding continuations at one level and passing them between other levels the user can essentially defeat this structuring, which is primitive only in the sense of being embedded in the treatment of READ-NORMALISE-PRINT (see section 5.c.iii). A diagram of these reflective levels was given in S6-58; the main idea is that code at each level is run by a processor which consists of its own small fragment of the structural field, and its own processor of one higher degree.

The crucial fact about each reflective level is that it is provided with its own processor state — its own environment and continuation structures. As will be explained in considerably more depth in the next section, the initial working assumption is that each level (except the lowest one) is initially running the code of the reflective processor; reflective procedures are *integrated* with this code in a very particular way. Since terms in

the reflective processor *designate* the program being processed (because of our association of designation and reflective levels), we can expect that the terms in user-provided reflective procedures will also designate fragments of the program being processed. In addition, other terms in the reflective processor designate the context in force for the processing of the program in the level below; again, reflective procedures will by and large do the same.

These assumptions are embodied in the formal protocols as follows. First, the parameter pattern of every reflective procedure is always bound to a normal-form designator of a sequence of three arguments: the argument structure of the reflective redex, the environment in effect at the point of normalisation of that redex, and the continuation ready to accept the result of that normalisation. Although the argument structure of the reflective redex will not have been normalised, reflective procedures (as did 2-LISP IMPRS) obey the basic principle that all bindings are in normal-form. In fact, as will become increasingly clear over the next pages, reflective procedures can be thought of as simple extensional procedures one level up. Thus the normal-form designator just mentioned will always be a three-element rail consisting of a handle (designating the argument structure), a rail (designating the environment), and a pair (a closure designating the continuation).

The environment in effect during the normalisation of the body of the reflective procedure will be the one that was in effect when the reflective procedure was defined, as always. Again, reflective procedures, except for the one fact that they shift levels upon being called, are otherwise entirely like simple procedures. The standard static scoping protocols apply as usual. However the *continuation* in effect will be the one mandated by the structure of the reflective processor, as explained in the next section.

In order to make all of this clear, we need to look at some simple examples. Since all reflective procedures are applied to three arguments, it is standard to use as the parameter pattern a three element rail. Thus suppose we were to begin defining a simple test procedure as follows:

```
(DEFINE TEST                                                    (S6-65)
   (LAMBDA REFLECT [ARGS ENV CONT]  ...  )
```

Note first the use of the function REFLECT in type position in the defining LAMBDA. REFLECT is in the same class as were EXPR and IMPR in 2-LISP (and SIMPLE in 3-LISP); it will be explained in section 5.b.iii below.

If the redex (TEST 1 2 3) were normalised in environment E_1 with continuation C_1, the atom TEST would first be looked up in E_1, and discovered to be bound to a reflective closure (the normal-form designator of a reflective procedure). The body of that closure would then be normalised in a context where ARGS was bound to a designator of the CDR of the reflective redex, ENV was bound to the normal-form designator of E_1, and CONT was bound to the normal-form designator of C_1. In particular, since the CDR of the reflective redex is the rail [1 2 3], ARGS would be bound to the handle '[1 2 3]. Similarly, environments are sequences of two-element sequences of atoms and bindings: since normal-form sequence designators are rails, ENV would be bound to a rail of two-element rails of the standard sort. Finally, CONT will be bound to the normal-form-designator of a continuation, which is a closure. About the procedural type of the closure nothing absolute can be said, for reasons that will become clear later. In the usual case, however, that closure will be a SIMPLE closure designed to accept a single argument — the local procedural consequence of the original redex (as is predicted by the continuations passed around in the meta-circular 2-LISP processor in section 4.d.vii — but see also section 5.c below).

These few introductory comments would lead us to expect the behaviour shown in the following console sessions. Note that the prompt character in 3-LISP has changed from that in 2-LISP: to the left of the caret is printed the index of the current reflective level. Since the various versions of TEST reflect up but do not come down again, each invocation causes the answer to be returned to the READ-NORMALISE-PRINT loop of the level above it (there are an infinite number of these loops all in effect simultaneously — this will all be explained in due course). In the first example we return simply the arguments, unmodified (we use (RETURN ARGS) in place of the simpler ARGS for a reason, unimportant here, that will emerge in the treatment of READ-NORMALISE-PRINT):

```
1> (DEFINE TEST1                                                (S5-66)
     (LAMBDA REFLECT [ARGS ENV CONT] (RETURN ARGS)))
1> TEST1
1> (TEST1 1 2 3)
2> '[1 2 3]
2> (TEST1)
3> '[]
```

The next example returns the environment rather than the arguments:

```
1> (DEFINE TEST2                                                    (S5-67)
     (LAMBDA REFLECT [ARGS ENV CONT] (RETURN ENV)))
1> TEST2
1> (LET [[X 3] [Y (+ 2 2)]] (TEST2 1 2 3))
2> [['X '3] ['Y '4] ... ]
```

Similarly, we can illustrate the return of the continuation, this time without defining a procedure but using a reflective LAMBDA form directly:

```
1> ((LAMBDA REFLECT [ARGS ENV CONT] (RETURN CONT)))                 (S5-68)
2> (<SIMPLE> ... )
```

As usual the primitive SIMPLE closure that is the CAR of all SIMPLE closures (including itself) is not printable, being circular; thus we will notate it as "<SIMPLE>" throughout.

So far, these examples are not very instructive. What is important about the *continuation* that is bound in each case to the atom CONT is that this is in fact the very continuation that was in effect when the reflective (TEST) redex was normalised. If it is called, the computation proceeding one level below will resume with the value passed to it by the explicit reflective procedure. For example, we can define a procedure called THREE that always calls its continuation with the numeral 3, irrespective of any arguments it is given:

```
1> (DEFINE THREE                                                    (S5-69)
     (LAMBDA REFLECT [ARGS ENV CONT]
       (CONT '3)))
1> THREE
1> (THREE)
1> 3
1> (+ 2 (THREE))
1> 5
1> (THREE (PRINT 'HELLO))          ; THREE ignores its arguments, without
1> 3                               ; normalising them.
```

When CONT is called in the body of THREE, the computation down one level proceeds, which results in the returning of a value to the top level of the level 1 version of READ-NORMALISE-PRINT in the fourth line of this example. Thus the numeral plays a standard role in that computation, as the example illustrates. Although the body of THREE was *itself* normalised at level 2, this fact is in some sense hidden from the *user* of the reflective procedure, since the reflect upwards was followed by a reflect downwards when the continuation was called.

This last fact is of considerable importance. In the previous examples using TEST, the reflective level was systematically increased, since each call to TEST returned to the level above it. This definition of THREE, however, since it *calls* CONT, although it *runs* one level

above that where it is used, does not *return* one level above. Thus procedures that are passed around and used in the normal way can be reflective procedures *without that fact needing to be noticed by their users.* Note also how much simpler the use of CONT was in these examples than the versions we toyed with in section 5.a. No STRIP was needed to awkwardly side-step the fact that we were binding ENV and CONT. Note as well that, since the 3-LISP processor is tail-recursive, no reflective continuations are saved in virtue of running THREE.)

As the last call to THREE illustrates, and as is evident from its definition, THREE ignores the arguments with which it was called. Furthermore, since THREE is reflective, those arguments are not normalised prior to being bound (ARGS in the last call to THREE would be bound to the handle '[(PRINT 'HELLO)]); therefore no potential side-effects take place.

Note as well that the call to CONT is given as an argument an expression that *designates the expression with which the continuation should proceed.* In our example, CONT is called with the handle '3 designating the numeral 3 — implying that the computation below should proceed *using that numeral.* In other words what is *mentioned* by the code making explicit use of the continuation is what is *used* by the code being processed. What is *explicitly used* in the reflective code (the continuation and environment, in particular), are *tacit* in the code being processed. Thus we have:

```
(TYPE 3)          ⇒   'NUMBER                                    (S6-70)
(TYPE '3)         ⇒   'NUMERAL
(TYPE (THREE))    ⇒   'NUMBER
```

From these last examples it may look as though forms are *de-referenced* by continuations, but it should be absolutely clear that this is not so. Rather, the difference in semantic level is a consequence of the difference in reflective level: *it is a difference of perspective on one and the same computation,* not a difference arising from some primitive act or event. It was true as well in the meta-circular processors for 1-LISP and 2-LISP: in processing the expression (+ 2 3), those processors manipulated *numerals,* not *numbers.* It is the same fact as our assumption throughout that Ψ is a function from s-expressions to s-expressions (nothing else would make sense). Furthermore, any attempt to violate this will cause an error:

```
1> (DEFINE THREE₂                                                    (S5-71)
     (LAMBDA REFLECT [ARGS ENV CONT]
       (CONT 3)))
1> THREE₂
1> (THREE₂)
TYPE-ERROR: CONT (at level 2), expecting an s-expression,
            was called with the number 3
```

Before we can make substantial use of these reflective abilities, we will need to introduce further machinery in the next sections. But we can construct some additional simple examples to illustrate the few points we have covered. First we define a rather vacuous procedure called VARIABLE so that any occurence of (VARIABLE <X>) (in an extensional context) will be entirely equivalent to a simple occurence of <X> on its own:

```
(DEFINE VARIABLE                                                     (S5-72)
   (LAMBDA REFLECT [[VAR] ENV CONT]
     (CONT (BINDING VAR ENV)))))
```

We have here used the recursive decomposition provided in parameter matching so that the parameter VAR will designate the *single* argument to VARIABLE. Thus for example, if we normalised (VARIABLE A), VAR would be bound to the handle 'A. Suppose in particular that we used VARIABLE as follows:

```
1> (LET [[A 3] [B (+ 2 2)]]                                          (S5-73)
     (+ (VARIABLE A) B))
1> 7
```

The redex (VARIABLE A) would be processed in the midst of normalising the argument expression for the extensional +. The environment in force at that point, which would be bound to ENV, would be:

```
[['A '3] ['B '4] ... ]                                               (S5-74)
```

Therefore the body of VARIABLE would be processed in an environment in which VAR was bound to 'A, ENV was bound to the rail given in S5-74, and CONT was bound to the continuation that was ready to accept an element of the arguments to +. The call to BINDING would explicitly look up A in that environment (BINDING was defined in S4-969), returning the handle '3. Thus the equivalent of the redex (CONT '3) would be processed, which would proceed the computation of +'s arguments appropriately.

Two things should be notable by their absence. First, in spite of the fact that the processing of S5-73 and the processing of the body of VARIABLE occur at different reflective levels, we did not need to avail ourselves of any explicit machinery to name or de-reference

expressions (no "↑" or "↓" appears in any of this code). As usual the semantic flatness ensures that everything works out correctly.

Second, there are no potential variable conflicts, since VAR and ENV are bound in a different environment from A and B. Thus we would have no trouble with:

```
1> (LET [[VAR 'HELLO]]                                          (S6-76)
       (RCONS (VARIABLE VAR) 'THERE))
1> '[HELLO THERE]
```

In this case the VAR in the pattern of VARIABLE would be bound to the handle 'VAR, and ENV would be bound to the environment designator

```
[['VAR ''HELLO] ... ]                                           (S6-76)
```

The body of VARIABLE would therefore *itself* be normalised in an environment of approximately the following form:

```
[['VAR 'VAR]                                                    (S6-77)
 ['ENV [['VAR ''HELLO] ... ]
 ['CONT <some continuation>]
 ... ]
```

However what is crucially important is that S6-76 is the level 1 environment, and S6-77 is the level 2 environment. The former is bound in the latter, but the two do not collide. This is exactly apropriate; the binding of ENV gives us access, and the separate environment gives us a place to stand.

5.b.ii. Some Elementary Examples

We turn next to some further examples that are almost as simple as the foregoing, but that are of some potential use. First, we can define a QUIT procedure, that returns the atom QUIT! as the result of an entire computation — that is, as the *result* of an explicit call to the tail-recursive normalising processor. The idea is to reflect once, and then simply to "return" the given atom. Since the reflective model of the interpreter is a "tail-recursive" program, a simple return will invoke the top level continuation *of the caller of the normaliser*, which will be the program that called this whole round of processing: namely, the READ-NORMALISE-PRINT loop. Thus we have:

```
(DEFINE QUIT                                                    (S6-78)
   (LAMBDA REFLECT ? 'QUIT!))
```

QUIT binds "?" (a regular atom that we will consistently use to indicate arguments we don't care about) to a rail designator of both arguments and context, and returns. Its precise behaviour can be better explained with reference to the READ-NORMALISE-PRINT code shown below in S6-194; what is relevant is that the atom QUIT! will be given to the top level of that code, which will print it out and the read in another expression for normalising. Thus our definition would engender the following behaviour:

```
1> (QUIT)                                                    (S6-79)
1> QUIT!
1> (+ 1 2)
1> 3
1> (+ 1 (/ (QUIT) 0))
1> QUIT!
1> [(PRINT 'HELLO) (QUIT) (PRINT 'THERE)] HELLO
1> QUIT!
```

Very similar to QUIT is the following definition of RETURN, which sends to the same caller of the processor a designator of an expression normalised in the RETURN redex's context (this is the RETURN we used in S5-66 through S5-68 above):

```
(DEFINE RETURN                                               (S5-80)
   (LAMBDA REFLECT [[EXP] ENV CONT]
      (NORMALISE EXP ENV ID)))
```

For example, if we were to process the following expression:

```
(LET [[X (- 3 3)]]                                           (S5-81)
    (NTH 2 [X (+ X X) (* X X) (RETURN X)]))
```

then when the RETURN redex was processed, it would reflect, binding EXP to 'X, ENV to [['X '0] ...], and CONT to a continuation that expected the final element for the normal-form rail being readied for NTH. The definition of RETURN, however, completely ignoring that continuation, normalises the argument in the context (thus obtaining the handle '0), and allows that result to return to the caller of the reflective procedure: the top level of this round of processing. Thus S5-81 would return a designator of the *numeral 0* to this top level; thus 0 would be printed out, as in:

```
1> (LET [[X (- 3 3)]]                                        (S5-82)
       (NTH 4 [X (+ X X) (* X X) (RETURN X)]))
1> 0
1> (RETURN (RETURN 4))
1> 4
```

These RETURNs are more useful when combined with procedures that can intercept their answers. Instead of RETURN such facilities are typically called CATCH and THROW in standard LISPs. We can define this kind of coordinated pair. The functionality we want is this (we will start simply): CATCH_1 will be a function of a single argument, whose result it merely passes back to its caller. However if somewhere within the dynamic scope of that argument there is an occurrence of a form (THROW_1 <EXP>), that result of normalising <EXP> will be returned straight away as the result of the entire CATCH_1 redex. Thus for example we would expect:

```
1> (DEFINE TEST                                                    (S6-83)
      (LAMBDA SIMPLE [X]
         (CATCH1                          ; Ready to accept a throw
            (+ (* X X)
               (/ X (IF (= X 3)
                        (THROW1 0)
                        (- X 3)))))))
1> TEST
1> (TEST 4)
1> 20                                     ; X of 4 works normally
1> (TEST 3)
1> 0                                      ; X of 3 exits prematurely
```

This sort of CATCH and THROW are trivially easy to define:

```
(DEFINE CATCH1                                                     (S5-84)
   (LAMBDA REFLECT [[ARG] ENV CONT]
      (CONT (NORMALISE ARG ENV ID))))

(DEFINE THROW1
   (LAMBDA REFLECT [[ARG] ENV CONT]
      (NORMALISE ARG ENV ID)))
```

The reason that these work is this: in the definition of CATCH, rather than giving NORMALISE the continuation that takes answers onto the final answer of the overall computation, the simple identity function is interposed between the result *of the argument of the* CATCH and the continuation with which the CATCH was called. Thus if that NORMALISE ever returns, the ID will flip the answer out to the explicit call to CATCH. Put another way, we have seen many times before that it is *argument* structure that embeds a process, not procedure calling. In general calls to NORMALISE are tail recursive, but the call to NORMALISE in S5-84 is crucially *not* tail-recursive: it very definitely embeds the processor one level. The definition of THROW shows how THROW returns the result of its argument *to the top of the current level of the computation*; since this will in general be the surrounding CATCH, the behaviour that we expected is simply generated.

Another common utility function of very much the same sort is UNWIND-PROTECT: a function of two arguments, such that the second argument is guaranteed to be processed after the first returns, no matter whether the first returns normally or directly (because of an error or RETURN or THROW or whatever). UNWIND-PROTECT can be defined as follows:

```
(DEFINE UNWIND-PROTECT                                              (S6-86)
   (LAMBDA REFLECT [[FORM1 FORM2] ENV CONT]
      (CONT (BLOCK1 (NORMALISE FORM1 ENV ID)
                    (NORMALISE FORM2 ENV ID)))))
```

where BLOCK1 is a form that processes an arbitrary number of arguments and returns the first:

```
(DEFINE BLOCK1                                                      (S6-86)
   (LAMBDA SIMPLE ARGS (1ST ARGS)))
```

Alternatively, UNWIND-PROTECT could be defined independently, as follows:

```
(DEFINE UNWIND-PROTECT                                              (S6-87)
   (LAMBDA REFLECT [[FORM1 FORM2] ENV CONT]
      (LET [[ANSWER (NORMALISE FORM1 ENV ID)]]
           (BLOCK (NORMALISE FORM2 ENV ID)
                  (CONT ANSWER)))))
```

Again, this definition succeeds because of the use of ID, and because the call to NORMALISE is *not* tail-recursive — rather, it is embedded in such a way that a full return to the continuation will be intercepted. The following definition, for example, would *not* work:

```
(DEFINE UNWIND-PROTECT                                              (S6-88)
   (LAMBDA REFLECT [[FORM1 FORM2] ENV CONT]
      (NORMALISE FORM1 ENV                          ; This definition
         (LAMBDA SIMPLE [FORM1!]                    ; would fail!
            (NORMALISE FORM2 ENV
               (LAMBDA SIMPLE [FORM2!] (CONT FORM1!)))))))
```

The problem here is that all of the subsequent intended processing is embedded in the continuation given to the normalisation of FORM1, and it is exactly this continuation which is neatly discarded by THROW and QUIT and so forth.

It should be observed that the use of BLOCK1 in S6-85 above is at the reflected level: thus the fact that BLOCK1 will normalise *its* argument (with the help of NORMALISE-RAIL) with a continuation is not problematic. It is not the *reflected level's continuation* that THROW and QUIT bypass; it is the continuation *passed around by the reflected level.*

As an example showing how THROW and CATCH interact smoothly with UNWIND-PROTECT, we have the following behaviour:

```
1> (DEFINE ADD-TO-X                                                    (S6-89)
     (LAMBDA SIMPLE [Y]
       (IF (= Y 0)
           (THROW X)                          ; ADD-TO-X increments X
           (BLOCK (SET X (+ X 1))             ; Y times, finally throwing
                  (ADD-TO-X (- Y 1))))))      ; X out as an answer.
1> ADD-TO-X
1> (DEFINE TEST
     (LAMBDA SIMPLE [Y]                       ; TEST saves X, and wraps
       (LET [[SAVE X]]                        ; protection around the
           (UNWIND-PROTECT (ADD-TO-X Y)       ; call to ADD-TO-X to
                           (SET X SAVE)))))   ; restore it on exit.
1> TEST
1> (SET X 3)
1> 3                                          ; Initialise X to 3
1> (CATCH (TEST 6))                           ; The THROW will come here.
1> 8                                          ; 5+3 is thrown out, but
1> X                                          ; X was restored between
1> 3                                          ; the THROW and the CATCH.
```

Similarly, UNWIND-PROTECT works correctly with the QUIT procedure defined earlier:

```
1> (UNWIND-PROTECT (BLOCK (SET X 100) (QUIT))                          (S6-90)
                   (SET X 4))
1> QUIT!
1> X
1> 4
```

The THROW and CATCH situation can be approached in quite a different fashion. It is standard, beyond the simple functionality we provided above, to define THROW and CATCH *tags* so that each CATCH identifies itself by name, and each THROW tosses a result to a named CATCH, rather than merely to the one closest in. One obvious approach would be for each THROW to return not just the intended result, but also the tag (in a two-element rail, say), and for each CATCH to check the identity of the tag, passing back the result if the tags matched, and proceeding in case they didn't. In particular, we could define such as pair as follows:

```
(DEFINE CATCH2                                                         (S6-91)
   (LAMBDA REFLECT [[TAG FORM] ENV CONT]
      (LET [[ANSWER (NORMALISE FORM ENV (LAMBDA SIMPLE X X))]]
          (IF (AND (SEQUENCE ANSWER) (= (LENGTH ANSWER) 2))
              (IF (= (1ST ANSWER) TAG)
                  (CONT (2ND ANSWER))
                  ANSWER)
              (CONT . ANSWER)))))

(DEFINE THROW2
   (LAMBDA REFLECT [[TAG EXP] ENV CONT]
      (NORMALISE EXP ENV
         (LAMBDA SIMPLE [EXP!] [TAG EXP!]))))
```

This then would support the following:

```
> (CATCH2 TAG1                                                    (S5-92)
     (+ 10 (CATCH2 TAG2
              (+ 20 (THROW2 TAG1 3)))))
> 3
> (CATCH2 TAG1
     (+ 10 (CATCH2 TAG2
              (+ 20 (THROW2 TAG2 3)))))
> 13
```

The definition of THROW_2 is straightforward; it is CATCH_2 that requires some explanation. If a thrown result is returned (recognised by the fact that a *two-element* result is returned: all standard results are single, as will be explained in section 5.d.i, below), then a check is made to see whether it was intended for this CATCH. If it was, then the thrown answer is given to CONT (implying that the THROW, so to speak, is stopped at this point); if it is not, then the answer is thrown back to the next embedding caller, etc.

More elegant than this approach, however, is the technique of binding the continuation to a particular name. The one requirement here is for dynamically scoped free variables (not unlike the problem we are concerned with, but more general — dynamic scoping is discussed below in section 5.d). Suppose in particular that we had a dialect where the redex

```
(DYNAMIC <ATOM>)                                                  (S5-93)
```

occurring in a pattern would bind <ATOM> *dynamically*, rather than statically, and where the same redex occuring in an extensional context would look it up dynamically. Thus for example if we had the following definitions:

```
(DEFINE SQUARE-ROOT                                               (S5-94)
   (LAMBDA SIMPLE [X] (SQRT-APPROX X 1)))

(DEFINE SQRT-APPROX                                               (S5-95)
   (LAMBDA SIMPLE [X ANS]
      (IF (< (ABS (- X (* ANS ANS)))
             (DYNAMIC ERROR))
          ANS
          (SQRT-APPROX X (/ (+ ANS (/ X ANS)) 2)))))
```

then we would expect the following behaviour (assuming we supported floating point arithmetic):

```
1> (LET [[(DYNAMIC ERROR) 0.1]]                                   (S5-96)
     (SQUARE-ROOT 2))
1> 1.417
```

We could then define CATCH and THROW as follows:

```
(DEFINE CATCH3                                                    (S6-97)
   (LAMBDA REFLECT [[TAG FORM] ENV CONT]
      (NORMALISE `(LET [[(DYNAMIC ,TAG) ,↑CONT]] ,FORM)
                 ENV
                 CONT)))

(DEFINE THROW3                                                    (S6-98)
   (LAMBDA REFLECT [[TAG FORM] ENV CONT]
      (NORMALISE FORM ENV
         (LAMBDA SIMPLE [FORM!]
            (NORMALISE `(DYNAMIC ,TAG) ENV
               (LAMBDA SIMPLE [CATCH-CONT] (↓CATCH-CONT FORM!)))))))
```

These are mildly awkward because they have to bind the continuation (which is at heart a reflected entity) in the dynamic environment *of the level below*, since it is that level's dynamic structure which is intended to control the scope of the tags.

In passing, the definition given above of SQUARE-ROOT rather inelegantly made SQRT-APPROX a globally available procedure. The following would be more discreet:

```
(DEFINE SQUARE-ROOT                                               (S6-99)
   (LABELS [[SQRT-APPROX
             (LAMBDA SIMPLE [X ANS]
                (IF (< (ABS (- X (* ANS ANS)))
                       (DYNAMIC ERROR))
                    ANS
                    (SQRT-APPROX X (/ (+ ANS (/ X ANS)) 2))))]]
      (LAMBDA SIMPLE [X]
         (SQRT-APPROX X 1))))
```

This works because of the fact that LABELS, as explained in chapter 4, uses our Z operator, making the definition of SQRT-APPROX appropriately recursive, and then gives the SQUARE-ROOT closure access to that recursive closure under the same name.

5.b.iii. LAMBDA, and Simple and Reflective Closures

We intend the semantics of 3-LISP's LAMBDA to remain unchanged from 2-LISP, although in the present dialect this function must be defined. Perhaps the simplest characterisation of LAMBDA is the following viciously circular definition (an alternative formulation was presented earlier as S4-475):

```
(DEFINE LAMBDA                                                    (S5-103)
   (LAMBDA REFLECT [[TYPE PATTERN BODY] ENV CONT]
      (REDUCE TYPE ↑[ENV PATTERN BODY] ENV CONT)))
```

Note, from the definition of REDUCE in for example S4-946, that this normalises the *referent* of TYPE; there is therefore no need for the following more complex version, which is behaviourally equivalent (in this way REDUCE differs from standard LISPS' APPLY):

```
(DEFINE LAMBDA                                                    (S5-104)
   (LAMBDA REFLECT [[TYPE PATTERN BODY] ENV CONT]
      (NORMALISE TYPE ENV                          ; This is equivalent
         (LAMBDA SIMPLE [TYPE!]                    ; to S5-103 above.
            (REDUCE TYPE! ↑[ENV PATTERN BODY] ENV CONT)))))
```

In sum, the first argument to LAMBDA is reduced with *designators* of the environment, pattern, and body. For example, if we were to normalise the following designator of the increment function:

```
(LET [[X 1]]                                                      (S5-105)
    (LAMBDA SIMPLE [Y] (+ X Y)))
```

the LAMBDA redex would be normalised in the following environment:

```
[['X '1] ... ]                                                    (S5-106)
```

LAMBDA would be called; the LAMBDA body would be normalised in the following (level 2) environment:

```
[['TYPE 'SIMPLE]                                                  (S5-107)
 ['PATTERN '[Y]]
 ['BODY '(+ X Y)]
 ['ENV [['X '1] ... ]]
 ['CONT (<SIMPLE> ... )]
 ... ]]
```

We may presume that the atom SIMPLE is bound in this environment to the primitive <SIMPLE> closure; thus TYPE will normalise to a designator of that closure. Thus the REDUCE redex in the last line of S5-103 is equivalent to the following (since we can substitute

bindings for variables in a flat language):

```
(REDUCE '<SIMPLE>                                              (S6-108)
        '[[['X '1] ... ] '[Y] '(+ X Y)]
        [['X '1] ... ]
        (<SIMPLE> ... ))
```

This is processed at level 2, but it is essentially the reflection of the following level 1 redex:

```
(<SIMPLE> [['X '1] ... ] '[Y] '(+ X Y))                        (S6-109)
```

Thus we see how the "current" environment is passed to <SIMPLE> (something we could not arrange except primitively in 2-LISP). We see as well how the inelegant level-crossing behaviour implied in our treatment of closures is indicated by the use of "↑" in S6-103.

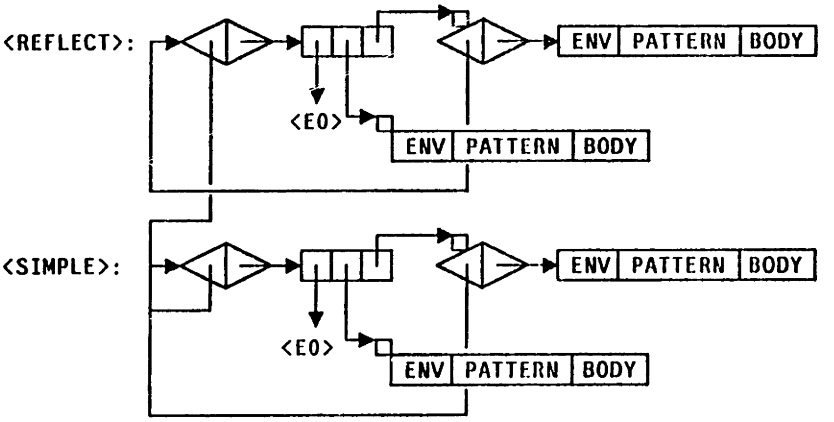
This treatment of LAMBDA puts the weight of lambda abstraction on the two primitive closure functions: <SIMPLE> and <REFLECT>. 3-LISP's <SIMPLE> is isomorphic to 2-LISP's <EXPR>: thus the <SIMPLE> closure would be notated as follows:

```
<SIMPLE> ≡  S: (:S :E0                                         (S6-110)
                   '[ENV PATTERN BODY]
                   '(:S ENV PATTERN BODY))
```

Similarly, we have the following structure to the primitive <REFLECT> closure (bound to the atom REFLECT in the initial environment):

```
<REFLECT> ≡  R: (<SIMPLE> :E0                                  (S6-111)
                          '[ENV PATTERN BODY]
                          '(:R ENV PATTERN BODY))
```

Like 2-LISP's <IMPR> and <MACRO> closures, 3-LISP's <REFLECT> is itself simple. These structures are notated graphically in the following diagram:



There is a consequence of these protocols that deserves to be made clear. Although a reflective procedure may run at one or other level, its own environment (the environment in which it was defined, and over which it was closed) is retained within it. Thus we can define a (highly inelegant) reflective version of INCREMENT as follows:

```
(DEFINE INCREMENT_R                                                    (S6-113)
   (LET [[X 1]]
      (LAMBDA REFLECT [[ARG] ENV CONT]
         (NORMALISE ARG ENV
            (LAMBDA SIMPLE [ARG!] (CONT ↑(+ X ↓ARG!)))))))
```

Note that X can of course not be added to ARG!, since ARG! will designate a *numeral*, not a *number*. Thus we would have:

```
1> (INCREMENT_R 3)                                                     (S6-114)
1> 4
1> (LET [[X (+ 2 3)]] (INCREMENT_R X))
1> 6
```

Even though the body of INCREMENT_R in this case will run at level 2, it will be normalised in the following environment:

```
[['ARG 'X]                                                             (S6-115)
 ['ENV [['X '6] ... ]]
 ['CONT (<SIMPLE> ... )]
 ['INCREMENT_R (<REFLECT> ... )]    ; The recursive binding provided by Z
 ['X '1]
 ... ]
```

Thus the binding of X will always be available within the body, no matter at what level INCREMENT_R is used. This is further indicated by showing the normal-form reflective closure to which INCREMENT_R is bound:

```
I: (<REFLECT> [['INCREMENT ':I] ['X '1] ... ]                          (S6-116)
              '[[ARG] ENV CONT]
              '(NORMALISE ARG ENV
                  (LAMBDA SIMPLE [ARG!] (CONT ↑(+ X ↓ARG!)))))
```

This whole closure is the closure that was elided in the fourth line of S6-115; in addition, the first argument in this closure is identical to the third tail of S6-115. Both of these facts follow from standard considerations of closures as explained in section c of the previous chapter.

Given this characterisation of how LAMBDA works, it is straightforward to define it.

As a first step, we can see straight away what the definition in S5-103 would reduce to. In particular, that definition would clearly bind LAMBDA to the following closure:

```
(<REFLECT> E0                                                    (S5-117)
           '[[TYPE PATTERN BODY] ENV CONT]
           '(REDUCE TYPE ↑[ENV PATTERN BODY] ENV CONT))))
```

Thus it might seem as if we could establish LAMBDA by executing the following:

```
(SET LAMBDA (<REFLECT> E0                                        (S5-118)
                       '[[TYPE PATTERN BODY] ENV CONT]
                       '(REDUCE TYPE ↑[ENV PATTERN BODY] ENV CONT)))))
```

However there are still three problems. First, this is still viciously circular because of the fact that REDUCE cannot be defined without first defining LAMBDA (since REDUCE is not primitive). Second, we have to discharge the "E0" in the reflective closure. Third, we also cannot use SET without defining it, which requires LAMBDA.

The first difficulty can be discharged by employing the up-down theorem: the closure demonstrated in S5-118 is provably equivalent to this:

```
(<REFLECT> E0                                                    (S5-119)
           '[[TYPE PATTERN BODY] ENV CONT]
           '(CONT ↑↓(PCONS TYPE ↑[ENV PATTERN BODY])))
```

which is the closure that would result (once LAMBDA were defined) from the following circular definition, which is equivalent to S5-103:

```
(DEFINE LAMBDA                                                   (S5-120)
   (LAMBDA REFLECT [[TYPE PATTERN BODY] ENV CONT]
      (CONT ↑↓(PCONS TYPE ↑[ENV PATTERN BODY]))))
```

The second problem (ridding the closure of E0) can be solved not by calling CURRENT-ENVIRONMENT, since we can't define it yet, but by inserting its body directly. The basic insight can be seen by noting that the following term will normalise to a designator of the environment in force when it is processed:

```
((LAMBDA REFLECT [[] ENV CONT] (CONT ↑ENV)))                     (S5-121)
```

Since we cannot use LAMBDA, we could equivalently write:

```
((REFLECT E0 '[[] ENV CONT] '(CONT ↑ENV)))                       (S5-122)
```

This would seem no better than S5-117, since E0 appears once again. However in S5-117 it is important that E0 be the real global environment (because REDUCE will be defined *later*, and we want that subsequent definition to be visible from the resulting closure); in S5-122

it needs only to support the processing of the body, which contains only three identifiers: CONT, NAME, and ENV. Two of these will be bound by the reflective pattern; thus we can merely construct an environment designator with the appropriate binding of NAME:

```
((REFLECT [['NAME ↑NAME]] '[[] ENV CONT] '(CONT ↑ENV)))                    (S5-123)
```

Inserting this into S5-122 then yields:

```
(<REFLECT> ((REFLECT [['NAME ↑NAME]]                                       (S5-124)
                     '[[] ENV CONT]
                     '(CONT ↑ENV)))
           '[[TYPE PATTERN BODY] ENV CONT]
           '(CONT ↑↓(PCONS TYPE ↑[ENV PATTERN BODY])))
```

Also, we can use REFLECT rather than <REFLECT>, since the CAR of this will be normalised when the whole is processed. Thus we wish to establish, as the initial binding of the atom LAMBDA, the result of normalising of the following term:

```
(REFLECT ((REFLECT [['NAME ↑NAME]]                                         (S5-125)
                   '[[] ENV CONT]
                   '(CONT ↑ENV)))
         '[[TYPE PATTERN BODY] ENV CONT]
         '(CONT ↑↓(PCONS TYPE ↑[ENV PATTERN BODY])))
```

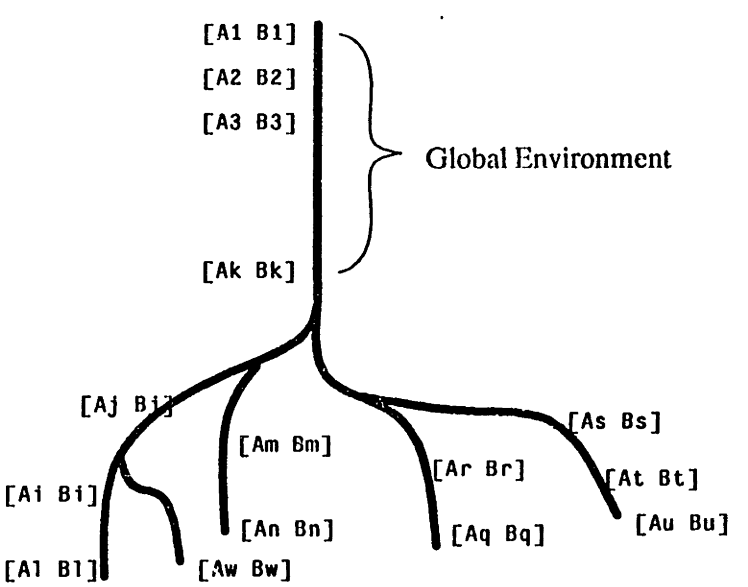
This is well defined, and indeed provides the behaviour we desire (in particular, S5-125 normalises to S5-119). The remaining third problem involves actually establishing it as LAMBDA's binding: we will not pursue that here, since it is merely tedious (since no LAMBDAS can be used in the process). We will merely take S5-125 as a reference definition for the moment, and assume that the binding has been established.

We say "for the moment" because there is in fact one remaining difficulty with S5-125, having to do with continuations, that makes its behaviour discernably different from that sketched in S5-103. We will ultimately *use* the definition in S5-125 to construct an improved version, in section 5.c.iv (see in particular S5-241). However the current version is sufficient for all the examples we will present in this chapter.

5.b.iv. The Structure of Environments

There is a question about environments, having to do with the extent to which environments are shared across levels. In 2-LISP we assumed that there was a single global environment — the primitively provided environment within the scope of which READ-NORMALISE-PRINT was called, so that all interaction with the processor through the communication channels took place with respect to this environment. In this environment thirty-two atoms were bound to the primitive closures, and so forth. Each LAMBDA form closed under this environment shared it in the way in which rails can share tails; in this way routines that worked side-effects onto environment designators could affect the environment in which previously defined procedures had been closed. Furthermore, destructive modification of otherwise unbound atoms caused the creation of bindings at the tail of this structure, making them maximally visible. In this way we were able to combine an entirely lexically scoped variable-scoping protocol with the provision of primitive routines that effected side-effects on structural field elements in such a way as to provide effective and convenient defining and debugging facilities for a programmer. In addition, as the discussion of recursion in section 4.c set forth, we were able to implement recursive definitions in terms of side-effects to the global environment. This structure is indicated in the following diagram:

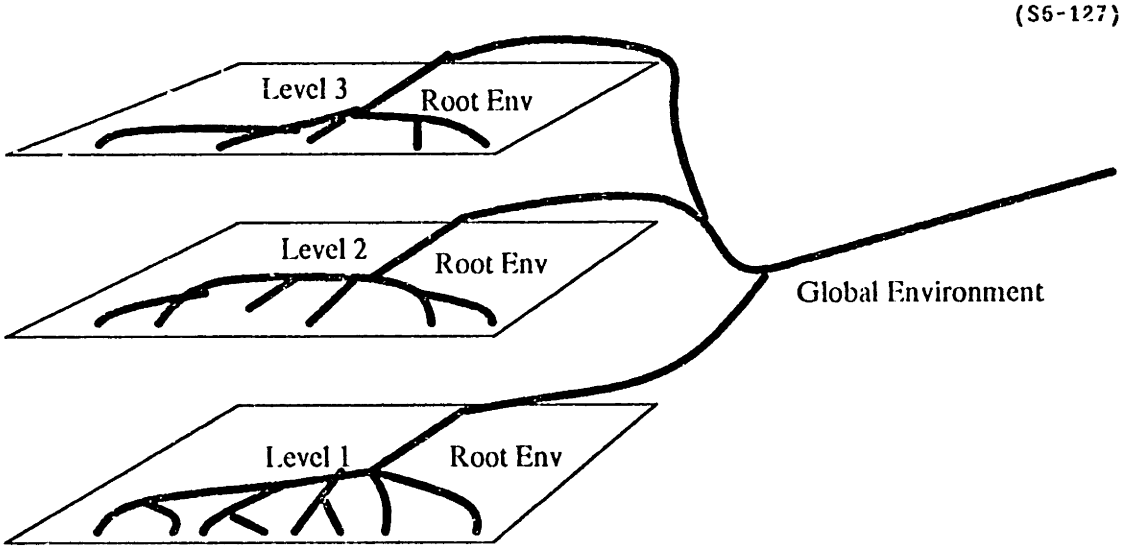
(S6-126)



In 3-LISP it is convenient as well to have a global environment, shared by each reflective level. If this were not the case, we would have to provide bindings for all of the primitive procedures at each level; when a new procedure was defined, it would have to be defined at each level if it were required at all levels, and so forth. The situation is not dissimilar to the situation that arises in a typed-logic, where different orders of predicates and logical particles are sometimes required at each type level. However, because each of our reflective levels is an untyped higher-order functional domain, we are assuming that no type considerations require differentiation among levels. We have in fact tacitly assumed this sharing in the examples already given: in S6-66, for example, we defined TEST_1 at level 1, but invoked it successfully at level 2. If the binding of the atom TEST_1 had not been established in a common context, the second invocation would have failed.

It will also prove convenient, however, to have what we will call a ***root*** environment for each level, global to all expressions within a given level, but private to that level. In this way we will be able to define special versions of procedures specific to a given level, without *necessarily* affecting all levels. The basic structure of this protocol is pictured as follows:

(S6-127)



Rather than fixing this arrangement inflexibly in the design of 3-LISP, however, we can instead introduce a rather more flexible arrangement that will allow this protocol to be used at will, as well as any other the user should define. This is because SET, as mentioned at the beginning of this section, is not a 3-LISP primitive. We said in chapter 4 that SET

could have been defined in terms of REBIND except that there was no way to provide the appropriate environment designator; in 3-LISP this problem is of course overcome. In particular, we assume the following (non-primitive) definition of REBIND (this is a simple 3-LISP version of S4-966 and S4-967):

```
(DEFINE REBIND                                                    (S5-128)
   (LAMBDA SIMPLE [VAR BINDING ENV]
      (IF (NORMAL BINDING)
          (REBIND* VAR BINDING ENV)
          (ERROR "Binding is not in normal form"))))

(DEFINE REBIND*                                                   (S5-129)
   (LAMBDA SIMPLE [VAR BINDING ENV]
      (COND [(EMPTY ENV) (RPLACT 0 ↑ENV ↑[[VAR BINDING]])]
            [(= VAR (1ST (1ST ENV)))
             (RPLACN 2 ↑(1ST ENV) ↑BINDING)]
            [$T (REBIND* VAR BINDING (REST ENV))])))
```

It is then straightforward to define a version of SET as follows (for the time being we will call this GSET, rather than SET, for reasons that will presently become clear):

```
(DEFINE GSET                                                      (S5-130)
   (LAMBDA REFLECT [[VAR BINDING] ENV CONT]
      (NORMALISE BINDING ENV
          (LAMBDA SIMPLE [BINDING!]
             (CONT (REBIND VAR BINDING! ENV))))))
```

As opposed to the situation in S4-695, where we had no appropriate binding of ENV, in the present circumstance the environment produced in virtue of the reflection is the correct argument to give to REBIND.

Finally, we define DEFINE in terms of GSET (again this is virtually identical to the 2-LISP version of S4-969, although we will define a non-primitive 3-LISP version of MACRO in section 5.d):

```
(DEFINE DEFINE                                                    (S5-131)
   (PROTECTING [Z]
      (LAMBDA MACRO [LABEL FORM]
         `(GSET ,LABEL
                (,↑Z (LAMBDA SIMPLE [,LABEL] ,FORM))))))
```

From none of these definitions, however, is the behaviour of so-called "global" bindings made obvious. In particular, we need to know the relationship between the "initial environment" and the environments with which each of the levels' READ-NORMALISE-PRINTS are called. Further, the question is one of deciding what is appropriate, since when we construct those levels in the next section we will be able to specify any behaviour we want.

The question of root environment arises most clearly in the case of global designators of arbtirary semantic entities, rather than in the specific case of function designators. We have seen already that having separate environments for each reflective level is hygenic, avoiding collisions and other confusions that would otherwise arise. It seems right to continue this separation — or at least to *enable* the user to continue it — for the establishing of names that transcend any particular local LAMBDA scope. What is required is to define SET not to search all the way into the global environment for bindings, but rather to establish the binding at the end of the root environment as appropriate. We call this set "LSET" for "level-SET", in distinction with the global "GSET". Such a definition (and a companion LEVEL-REBIND) can be defined as follows:

```
(DEFINE LSET                                                    (S6-132)
   (LAMBDA REFLECT [[VAR BINDING] ENV CONT]
      (NORMALISE BINDING ENV
          (LAMBDA SIMPLE [BINDING!]
             (CONT (LEVEL-REBIND VAR BINDING! ENV))))))

(DEFINE LEVEL-REBIND                                            (S6-133)
   (LAMBDA SIMPLE [VAR BINDING ENV]
      (IF (NORMAL BINDING)
          (LEVEL-REBIND* VAR BINDING ENV)
          (ERROR "Binding is not in normal form"))))

(DEFINE LEVEL-REBIND*                                           (S6-134)
   (LAMBDA SIMPLE [VAR BINDING ENV]
      (COND [(EMPTY ENV) (RPLACT 0 ↑ENV ↑[[VAR BINDING]])]
            [(= VAR (1ST (1ST ENV)))
             (RPLACN 2 ↑(1ST ENV) ↑BINDING)]
            [(= (REST ENV) GLOBAL)
             (RPLACT 0 ENV
                     (PREP* (1ST ENV) ↑[[VAR ↑BINDING]] (REST ENV)))]
            [$T (REBIND* VAR BINDING (REST ENV))])))
```

PREP* is a multi-argument version of PREP defined as follows:

```
(DEFINE PREP*                                                   (S6-135)
   (LAMBDA SIMPLE ARGS
      (COND [(EMPTY ARGS) (ERROR "Too few arguments to PREP*")]
            [(UNIT ARGS) (1ST ARGS)]
            [$T (PREP (1ST ARGS) (PREP* . (REST ARGS)))])))
```

The S6-132 definition of LSET will engender the expected behaviour just in case READ-NORMALISE-PRINT is called with an environment which has the global environment as a tail, but it not itself identical with that global environment. The protocols we adopt in section 5.c will have this property; thus we will assume LSET and GSET in subsequent examples. In addition, since we always use DEFINE to define procedures, which we have defined in terms

of GSET, we will therefore use the term SET (which remains still unused) as an alias of LSET. In other words we will assume:

```
(DEFINE SET LSET)                                                    (S6-136)
```

Thus procedures will by default be globally accessible; variables set in virtue of SET and LSET will be accessible only on a level-specific manner. Truly global variables should be set using GSET explicitly.

It should be realised that these are only conventions: they are not part of the 3-LISP definition, but a protocol we will find convenient for subsequent examples. In addition, it should be clear that closures — even those that themselves may run at any level — will be closed in the environment that includes the root environment of their place of definition. We will illustrate this with a highly inelegant example. First, we define a test reflective procedure called UP that returns to the READ-NORMALISE-PRINT of some level above it:

```
(DEFINE UP                                                           (S6-137)
   (LAMBDA REFLECT [[ARG] ENV CONT]
      (NORMALISE ARG ENV
         (LAMBDA SIMPLE [ARG!]
            (IF (= ↓ARG! 1)
                (RETURN 'OK)
                (UP (- ↓ARG! 1)))))))
```

Thus we would expect the following behaviour:

```
1> (UP 3)                                                            (S6-138)
4> 'OK
4> (UP 2)
6> 'OK
```

Then suppose we define a level-specific variable X, and define a procedure on this level (which will thus have access to it):

```
1> (LSET X 100)                    ; Give X a level-                 (S6-139)
1> X                               ; specific value of 100
1> (DEFINE TEST                    ; and define TEST to
      (LAMBDA SIMPLE [Y] (+ X Y)))  ; use X freely.
1> TEST
1> X
1> 100                             ; X is 100 here at level 1
1> (TEST 3)
1> 103                             ; TEST adds 100
1> (UP 6)                          ; Move up to level 6
6> 'OK
6> X                               ; X is not bound at level 6
ERROR at level 6: X is unbound
6> (TEST 3)
6> 103                             ; But TEST still adds 100
```

```
6> (SET X 20)                    ; We can give X a level 6
6> X                             ; value of 20.
6> (TEST X)                      ; Now X is bound to 20, but
6> 120                           ; TEST continues to add 100.
```

The procedure TEST adds the level 1 value of 100, since it is closed in that environment, no matter where it is used. The name TEST is made globally available (by DEFINE) as usual, but the TEST closure remains defined in that level 1 environment, as the examples show. If, however, we return to level 1 and reset X, as in the following, then TEST is modified at that and at every other level:

```
1> (SET X 5)                     ; Come back to level 1          (S6=140)
1> X                             ; and reset X.  Now
1> (TEST 3)                      ; TEST adds 5, rather than
1> 8                             ; 100, here or at any other
1> (UP 4)                        ; level.
5> 'OK
5> (TEST 20)
5> 25
```

What are we to conclude from these examples (which are hardly elegant)? The answer is this: environments are by and large independent of reflective level: the whole amalgam of lexical scoping protocols, closures, and the rest (as we have seen in the previous chapters) make the environment structure of a process leafy and shallow, and quite orthogonal to the continuation structure, which more accurately represents the recursive descent of the procedures being called. The simplest solution to the problem of how environments interact with reflective levels, then, is this: they do not. Reflection has to do more with mention of programs, and with independent continuations, than it does with independent environments, *since, in a statically scoped dialect, environments are kept by and large independent from one closure to the next.* However what the previous examples have illustrated is that we can extend this basic position so as to allow some level-specific environment, without the need for more primitives. We will rarely depend on level-specific bindings, but from time to time they will prove convenient.

As a final footnote, we should observe that the use of X in the manner of TEST in S5-139 is far from recommended practice. Much more reasonable is to give TEST its own copy of a binding of X, as in (we demonstrated this kind of definition in section 4.c):

```
(DEFINE TEST                                                    (S6-141)
   (LET [[X 100]]
      (LAMBDA SIMPLE [Y] (+ X Y))))
```

The resultant TEST could be used at any reflective level, as usual. Should the private version of X ever need to be changed, we would do so using an explicit REBIND on the environment contained in TEST's resultant closure, as follows:

```
(REBIND 'X '6 (ENV ↑TEST))                                    (S6-142)
```

This is all far simpler, and far more elegant, than the unhappy behaviour of S6-139 and S6-140.

5.b.v. Simple Debugging

One place that reflection is likely to prove useful is as an aid to debugging. The 3-LISP reflective protocols are not themselves debugging protocols, but it is simple enough to build such behaviour on top of them. We will look at some simple suggestions in this section. In section 5.d we sketch various ways in which interrupts might be connected to the reflective machinery, but we will restrict ourselves here to situations in which a program itself recognises that a trouble has arisen, and makes an explicit call to an error package.

Suppose for example we were to define a procedure such as the following, with a call to a procedure called DEBUG (this assumes a version of + that accepts an arbitrary number of arguments):

```
(DEFINE AVERAGE                                                    (S5-147)
   (LAMBDA SIMPLE [SEQ]
      (IF (EMPTY SEQ)
          (DEBUG "AVERAGE was called with an empty sequence")
          (/ (+ . SEQ) (LENGTH SEQ)))))
```

Our intent is to have DEBUG interact with the user, by printing out the message, and allowing access to the computation that was in force. We expect to support, in particular, something like the following scenario:

```
1> (SET X [1 3 5 7 9])                                             (S5-148)
1> [1 3 5 7 9]
1> (AVERAGE X)
1> 5
1> (DEFINE SUM-AND-AVERAGE
      (LAMBDA SIMPLE [S]
         [(+ . S) (AVERAGE S)]))
1> SUM-AND-AVERAGE
1> (SUM-AND-AVERAGE X)
1> [25 5]
1> (SET Y [])
1> Y
1> (SUM-AND-AVERAGE Y)
ERROR: AVERAGE was called with an empty sequence
2>
```

It seems natural that DEBUG should interact with the user at level 2, although this will be revised later. At this point we expect the user to be able to test the bindings of various variables. In particular, suppose we are interested in the binding of SEQ. We cannot use this name directly:

```
2> SEQ                                                          (S5-149)
ERROR: SEQ unbound variable
```

Rather, it must be looked up in the environment that was made available to DEBUG (we assume DEBUG is a reflective procedure). Suppose for example that DEBUG inelegantly SET the atoms ENV and CONT *at its level* (we will use LSET) and then returned the error message (this is an awkward definition that we will soon replace):

```
(DEFINE DEBUG                                                   (S5-150)
   (LAMBDA REFLECT [[MESSAGE] ENVIRONMENT CONTINUATION]
      (BLOCK (LSET ENV ENVIRONMENT)
             (LSET CONT CONTINUATION)
             (RETURN MESSAGE))))
```

(There are problems with the message part of this, but we will ignore them for now.). Then we might expect the following (as a continuation of S5-149):

```
2> (BINDING SEQ ENV)                                            (S5-151)
ERROR: SEQ unbound variable
2> (BINDING 'SEQ ENV)
2> '[]
```

In other words we need expressly to look up the binding in the "globally" bound ENV, where the appropriate empty sequence is found.

Suppose we decide to materially alter SEQ to be a sequence of three integers — not only the binding of SEQ, but the rail to which it was bound (in other words we intend to affect the binding of Y as well); we would perform the following:

```
2> (RPLACT 0 (BINDING 'SEQ ENV) '[-5 0 20])                     (S5-152)
2> '[-5 0 20]
```

We check to make sure our alteration took effect.

```
2> (BINDING 'SEQ ENV)                                           (S5-163)
2> '[-5 0 20]
```

Had we wanted only to change the parameter binding in AVERAGE, we could instead have used:

```
2> ENV                                                          (S5-164)
2> [['SEQ '[]] ... ]
2> (RPLACN 2 (1ST ENV) '[-5 0 20]))
2> ['SEQ '[-5 0 20]]
2> (BINDING 'SEQ ENV)
2> '[-5 0 20]
```

Or, using the REBIND of S5-128:

```
2> (REBIND 'SEQ '[-5 0 20] ENV)                                    (S5-165)
2> 'SEQ
2> (BINDING 'SEQ ENV)
2> '[-5 0 20]
```

No matter how this is done, we are set with SEQ bound to a non-empty rail. Suppose we now want to continue the computation. The first task is to obtain access to the appropriate continuation. CONT was bound by DEBUG; we can try to look it up in ENV:

```
2> (BINDING 'CONT ENV)                                             (S5-166)
ERROR: CONT unbound variable
```

But of course CONT is bound *at the reflective level*, since it is a theoretical entity, not part of the code being debugged:

```
2> CONT                                                            (S5-167)
2> (<SIMPLE> [...] ... )
```

Suppose now that we tried to use it, returning the result that would have been engendered had SEQ been bound to [-5 0 20] all along:

```
2> (CONT (/ (+ . SEQ) (LENGTH SEQ)))                               (S5-168)
ERROR: SEQ unbound variable
```

This error is of course to be expected: once again we are attempting to use a variable at the *current* level, when it belongs one level below. An inelegant attempted repair is this:

```
2> (LET [[S2 (BINDING 'SEQ ENV)]]                                  (S5-169)
      (CONT (/ (+ . S2) (LENGTH S2)))
TYPE-ERROR: +, expecting a number, found the numeral '-5
```

Once again the level problem intervenes: S2 is bound to a *designator* of the binding of SEQ. This too we could try to circumvent:

```
2> (LET [[S2 (BINDING 'SEQ ENV)]]                                  (S5-160)
      (CONT (/ (+ . ↓S2) (LENGTH ↓S2)))
TYPE-ERROR: CONT, expecting an s-expression, found the number 6
```

Again a type error: CONT was bound to a continuation that expected the *designator* of the average; not the average itself. A final fix in this terrible direction is this:

```
2> (LET [[S2 (BINDING 'SEQ ENV)]]                                  (S5-161)
      (CONT ↑(/ (+ . ↓S2) (LENGTH ↓S2)))
1> [0 6]
```

However all of this ugliness is telling us something. The point is that we have tried to execute at a reflective level a computation that was intended to be executed at the base level one below us. Certainly a far better treatement would be the following:

```
2> (NORMALISE '(/ (+ . SEQ) (LENGTH SEQ)) ENV CONT)                (S6-162)
1> [0 5]
```

Ths is simpler, and is semantically reasonable. The appropriate variables are set, used in the appropriate environments, and the correct continuation is supplied, yielding a final answer at level 1.

Of course as an answer, `[0 5]` is incorrect, since the *sum* was performed over SEQ while it was still bound to the null sequence `[]`, whereas the average was performed over the new binding of SEQ to `[-5 0 20]`. In practice one would want to redo the whole computation, or use more sophisticated continuation examining functions of the sort described in section 5.d.

Now that we have returned to base level, we can see the differences in how we changed SEQ; if we executed S6-152 we would now have:

```
1> Y                                                               (S6-163)
1> [-5 0 20]                              ; Y was altered
```

If on the other hand we had chosen S6-154 or S6-155, Y would remain bound to the same null sequence:

```
1> Y                                                               (S6-164)
1> []                                     ; Y is unchanged
```

There is an undeniable price paid for the strict separation in environments maintained between reflective levels, and an argument can be mounted that it would be more convenient to interact with a READ-NORMALISE-PRINT loop *at level 1*, rather than at the reflected level looking down. In addition, we rather inelegantly had to use LSET to set level variables ENV and CONT in order to make them available to the user. These realisations suggest a different approach. Suppose that instead of S6-150 we had defined DEBUG as follows:

```
(DEFINE DEBUG                                                      (S6-166)
   (LAMBDA REFLECT [[MESSAGE] ENV CONT]
      (BLOCK (TERPRI)
             (PRINT MESSAGE)
             (READ-NORMALISE-PRINT ENV))))
```

This is a good start, but DEBUG will be useful only if there is some way in which to *leave* the READ-NORMALISE-PRINT, which is otherwise an infinite computation. We can always reflect out of it, and thereby return from the BLOCK, but that would return to the top level of the level 1 computation, which is not what we intend. What is striking about this definition, however, is that the RETURN we have already defined will suffice, if we merely modify S6-166 as follows:

```
(DEFINE DEBUG                                                    (S6-166)
   (LAMBDA REFLECT [[MESSAGE] ENV CONT]
      (BLOCK (TERPRI)
             (PRINT MESSAGE)
             (CONT (READ-NORMALISE-PRINT ENV)))))
```

Given this last definition we could have the following session (note that all user interaction is at level 1, in spite of level 2 machinations going on over its head):

```
1> (SET 'Y [])                                                   (S6-167)
1> []
1> (TEST Y)
AVERAGE was called with an empty sequence
1> SEQ                                  ; We can use SEQ directly
1> []
1> (SET 'SEQ [-6 0 20])                 ; Similarly we set it at this
1> [-6 0 20]                            ; level.
1> (/ (+ . SEQ) (LENGTH SEQ))           ; This is the correct average,
1> 6                                    ; But it simply prints it.
1> (RETURN (/ (+ . SEQ) (LENGTH SEQ)))  ; Call RETURN with this, and
1> [0 6]                                ; the computation completes.
1> Y
1> []                                   ; Note that Y remains null.
```

What is crucial to understand about S6-167 is that the fourth through eleventh lines are interactions with the *reflectively embedded* call to READ-NORMALISE-PRINT. A better user protocol would be to use a variant on READ-NORMALISE-PRINT that prints a distinguishable prompt character that indicates that the user interaction remains at level 1, but that the call is embedded. Something of the following sort is indicated:

```
1> (SET 'Y [])                                                   (S6-168)
1> []
1> (TEST Y)
AVERAGE was called with an empty sequence
1>> SEQ                                 ; We can use SEQ directly
1>> []
1>> (SET 'SEQ [-6 0 20])                ; Similarly we set it at this
1>> [-6 0 20]                           ; level.
1>> (/ (+ . SEQ) (LENGTH SEQ))          ; This is the correct average,
1>> 6                                   ; But it simply prints it.
1>> (RETURN (/ (+ . SEQ) (LENGTH SEQ))) ; Call RETURN with this, and
```

```
1> [0 6]                                  ; the computation completes.
1> Y
1> []                                     ; Note that Y remains null.
```

What is important about this example is the recognition that reflective level procedures facilitate the debugging protocols substantially, but that user interaction *at the reflected level* was quite inconvenient. It proved much easier to interact with the errant code in an embedded READ-NORMALISE-PRINT loop at the same level as the bug, rather than above it. It is exactly this sort of recognition that 3-LISP can facilitate — without both the semantic rationalisation of 2-LISP and the reflective abilities of 3-LISP we would not have been able even to ask the question, let alone come upon an answer in this simple way.

5.b.vi. REFERENT

In section 4.d.i we defined the 2-LISP version of REFERENT, noting that it inherently mandated a second normalisation (of the referent of its argument expression), and that that normalisation took place in the context that resulted from the standard normalisation of its primary argument. We commented as well that this was perhaps inappropriate; that it would be reasonable to require of REFERENT that a *second* argument be provided that designated the context in which the referent of the first argument was to be processed. In 3-LISP we will adopt this suggestion, since context designators are of course straightforwardly obtainable.

We will require, in other words, REFERENT redexes of the following form:

```
(REFERENT <EXP> <ENV>)                                              (S5-169)
```

where <EXP> is taken to designate an expression, and <ENV> an environment, and where the whole redex designates the referent of the expression designated by <EXP> in the environment designated by <ENV>. Thus for example we would have:

```
(REFERENT '3 [])                    ⇒   3                           (S5-170)
(REFERENT 'X [['X '4]])             ⇒   4
(LET [[A 'B]
      [B 4]
      [ENV [['A '5]['B '6] ... ]]]
   (REFERENT A ENV)                 ⇒   6
```

The last example illustrates how the environment used to establish what expression and what environment are intended, and the environment used to establish the subsequent referent, play different roles.

Though this protocol is general and acceptable, there is something odd about it, which should be brought out. REFERENT (and NAME), as we have said all along, involve a kind of *level-crossing* behaviour that is rather different in flavour from the kind of behaviour mandated by reflective procedures. As we have been at pains to indicate, reflective procedures don't so much *shift* the level of the processor; rather, they are procedures that are *run* at a different level than that in which the reflective redex occurs, but that upper reflected level is considered always to exist — running the reflective procedure correctly amounts merely to integrating it into the level as appropriate.

Therefore the use of REFERENT in a reflective procedure is only occasionally indicated. One good example is the definition of UP given above in S5-137; our definition there was as follows:

```
(DEFINE UP                                                    (S5-171)
   (LAMBDA REFLECT [[ARG] ENV CONT]
      (NORMALISE ARG ENV
         (LAMBDA SIMPLE [ARG!]
            (IF (= ↓ARG! 1)
                (RETURN 'OK)
                (UP (- ↓ARG! 1)))))))
```

Much more perspicuous, however, is the following:

```
(DEFINE UP                                                    (S5-172)
   (LAMBDA REFLECT [[ARG] ENV CONT]
      (LET [[N (REFERENT ARG ENV)]]
         (IF (= N 1)
             (RETURN 'OK)
             (UP (- N 1))))))
```

This code binds N to the actual result of normalising the argument to UP *in the environment of the original* UP *redex*, rather than to a *designator* of it, which is what ARG! was bound to in S5-171. However there are many cases — the last line of S5-171 is one — where there is no motivation to supply a different environment than the one covering the redex as a whole. One such case (this is the circumstance in S5-171) arises when the argument to REFERENT is known to be in normal-form, and hence the second argument to REFERENT is immaterial. We have not explained what "↓ARG!" expands to in 3-LISP; the current discussion indicates that it will be some form (REFERENT ARG! <ENV>). The fact that ARG! designates a normal-form expression implies that it will not matter what the second argument is, in the particular case we are considering.

Our general policy has been to align reflective level and semantic level — NAME and REFERENT are provided to allow additional flexibility, since we are allowing each reflective level its own meta-structural powers, above and beyond those implied in the very architecture of the reflective hierarchy. We will therefore arrange it so that it is convenient to use the current environment as an explicit second argument to REFERENT, and will make this the normal expansion of ↓. In particular, we can define a procedure called CURRENT-ENVIRONMENT as follows (note the use of [] as the argument pattern, implying that CURRENT-ENVIRONMENT must be called with no arguments):

```
(DEFINE CURRENT-ENVIRONMENT                                    (S6-174)
   (LAMBDA REFLECT [[] ENV CONT] (CONT ↑ENV)))
```

The use of NAME (in the "↑") makes manifest the fact that CURRENT-ENVIRONMENT embodies a fundamental level-shifting kind of operation: giving a designator of the environment to code at that very level. We will then simply posit that the lexical notation using the down-arrow will use this procedure. It would be simple to define the notational expansion as follows:

```
↓<EXP>    ≡>    (REFERENT <EXP> (CURRENT-ENVIRONMENT))          (S6-175)
```

However this relies (since it is a *macro* expansion) dangerously on the bindings of the atoms CURRENT-ENVIRONMENT and REFERENT. Thus we will instead adopt the following normal-form version of the same thing (the atoms have been replaced with the closures that S6-175 assumes they are bound to):

```
↓<EXP>    ≡>    (<primitive-REFERENT-closure>                  (S6-176)
                 <EXP>
                 ((<REFLECT> E0
                             '[[] ENV CONT]
                             '(CONT (<primitive-NAME-closure> ENV))))
```

This will successfully deal with both problems: meta-structural operations that intend to remain within a given level, and the explicit de-referencing of expressions in normal-form, where the environment argument makes no difference.

There is one final remark to be made about REFERENT, having to do with continuations. We have said nothing about what continuation is used for the second normalisation indicated by a REFERENT redex, but we can retain the answer we provided in 2-LISP (as indicated in S4-949); namely: the same one given to the REFERENT redex as a whole. In other words, if (REFERENT '(1ST [1 2 3]) ENV) were normalised in environment

E_1 with continuation C_1, then the handle '(1ST '[1 2 3]) and the atom ENV would be normalised in environment E_1, with some other continuation C_2 (the exact form of C_2 will be demonstrated in section 5.c). Suppose that the atom ENV designates (in E_1) an environment that we call E_2 in our meta-language. Then the structure designated by the handle '(1ST [1 2 3]) — namely, the redex (1ST [1 2 3]) — will be normalised in environment E_2 *and with continuation* C_1. REFERENT, in other words, normalises the referent of its argument expression *tail-recursively*. This fact is made evident in the fifth line of the definition of MAKE-C1 in the listing of the reflective processor given in S6-207 in section 5.c.

5.b.vii. The Conditional

We said at the outset that 3-LISP provides an extensional conditional called EF, in place of an intensional IF. We can define IF in terms of EF: the approach is to use reflection to obtain proper access to the appropriate contexts, and to use the intensionality of LAMBDA in the standard way to defer processing. Ignoring for a moment the question of reducing IF with non-rail CDRs, we have:

```
(DEFINE IF1                                                    (S5-176)
   (LAMBDA REFLECT [[PREMISE C1 C2] ENV CONT]
      (NORMALISE PREMISE ENV
         (LAMBDA SIMPLE [PREMISE!]
            ((EF (= PREMISE! '$T)
                 (LAMBDA SIMPLE [] (NORMALISE C1 ENV CONT))
                 (LAMBDA SIMPLE [] (NORMALISE C2 ENV CONT))))))))
```

The crucial aspect of this definition is the fact that the second and third arguments to EF (which is processed at the reflected level) are LAMBDA terms, rather than simply NORMALISE redexes. Thus, in the processing of the EF redex, two closures will be produced, since EF is procedurally extensional, but only one of them will be returned as the result of the EF redex. That one result is then reduced with no arguments (this is why there are two parentheses to the left of the "EF" atom in the fifth line). Thus, if the premise normalises to $T, then the first closure will be reduced; otherwise the second. Since it is only on reduction of the constructed closures that the consequents are normalised, we thus have the appropriate behaviour. In particular, whereas EF would yield the following:

```
1> (EF (= 1 2)                                                 (S5-177)
       (PRINT 'YES)
       (PRINT 'NO)) YES NO
1> $T
```

the intensional IF would on the other hand yield:

```
1> (IF1 (= 1 2)                                                    (S6-178)
       (PRINT 'YES)
       (PRINT 'NO)) NO
1> $T
```

This works because the C1 and C2 parameters in the definition of IF_1 in S6-176 would be bound, respectively, to the handles '(PRINT 'YES) and '(PRINT NO). The EF redex would effectively be of the form

```
(EF (= '$F '$T)                                                    (S6-179)
    (LAMBDA SIMPLE [] (NORMALISE '(PRINT 'YES) ENV CONT))
    (LAMBDA SIMPLE [] (NORMALISE '(PRINT 'NO)  ENV CONT)))
```

which would normalise to the following closure of the third line:

```
(<SIMPLE> [ ... ] '[] '(NORMALISE '(PRINT 'NO) ENV CONT))          (S6-180)
```

When this is reduced with a null argument, the body would be normalised, causing the processing of the second consequent, as expected.

The only additional subtlety to consider is the use of non-rail CDRs. Since EF is extensional we have no trouble in its case:

```
1> (EF . (REST [(= 1 2) (= 2 2) (+ 1 2) (+ 2 2)]))                 (S6-181)
1> 3
1> (MAP EF [(= 1 1) (= 1 2) (= 1 3)]
           [(+ 1 1) (+ 1 2) (+ 1 3)]
           [(* 1 1) (* 1 2) (* 1 3)])
1> [2 2 3]
```

On the other hand neither of the expressions using EF in S6-181 would work using IF (assuming the definition of MAP of S4-991 was carried over from 2-LISP):

```
1> (IF . (REST [(= 1 2) (= 2 2) (+ 1 2) (+ 2 2)]))                 (S6-182)
ERROR: Bad pattern match
1> (MAP IF [(= 1 1) (= 1 2) (= 1 3)]
           [(+ 1 1) (+ 1 2) (+ 1 3)]
           [(* 1 1) (* 1 2) (* 1 3)])
ERROR: Bad pattern match
```

We could, as initially suggested in S4-398, complicate the definition of IF as follows:

```
(DEFINE IF2                                                        (S6-183)
   (LAMBDA REFLECT [ARGS ENV CONT]
      (LET [[[PREM C1 C2]
             (IF (RAIL ARGS)                    ; This will not do.
                 ARGS
                 (NORMALISE ARGS ENV ID))]]
         (NORMALISE PREM ENV
```

```
        (LAMBDA SIMPLE [PREM!]
           ((EF (= PREM! 'ST)
                (LAMBDA SIMPLE [] (NORMALISE C1 ENV CONT))
                (LAMBDA SIMPLE [] (NORMALISE C2 ENV CONT)))))))))
```

However this cannot stand, since there is a viciously circular use of IF within the body. When we were meta-circularly defining IF in S4-398 this didn't matter, but here we are actually proposing a definition that is intended to be self-sufficient. Though it might seem possible to replace the inner IF with an EF, that would always normalise all three arguments, so it is not an answer.

There is a solution, however: we can iterate our technique of avoiding processing by wrapping expressions in LAMBDAS, as follows:

```
(DEFINE IF3                                                          (S5-184)
  (LAMBDA REFLECT [ARGS ENV CONT]
    ((EF (RAIL ARGS)
         (LAMBDA SIMPLE []
           (LET [[[PREMISE C1 C2] ARGS]]
              (NORMALISE PREMISE ENV
                 (LAMBDA SIMPLE [PREMISE!]
                   ((EF (= PREMISE! 'ST)
                        (LAMBDA SIMPLE [] (NORMALISE C1 ENV CONT))
                        (LAMBDA SIMPLE [] (NORMALISE C2 ENV CONT)))))))))
         (LAMBDA SIMPLE []
           (NORMALISE ARGS ENV
              (LAMBDA SIMPLE [[PREMISE C1 C2]]
                 (CONT (EF (= PREMISE! 'ST) C1 C2)))))))))
```

In other words, if the argument expression to an IF_3 is a rail, then the premise expression (the first element of the rail) is normalised, and depending on its result either one or other of the consequents (the second and third elements of the rail) are normalised *tail-recursively* (this is important). If the argument expression is *not* a rail, it is normalised as a unit; the continuation destructures it into the appropriate pieces, returning whichever piece is appropriate (no further processing is required in this case, of course).

We will take the definition in S5-184 as our reference. It should be noted, however, that if it were not for the ability to handle non-rail CDRS, the simpler definition in S5-176 would suffice. On the other hand, if that simpler behaviour were considered acceptable, we could use the following even simpler macro (once we define MACRO in section 5.d.ii):

```
(DEFINE IF4                                                          (S5-185)
   (LAMBDA MACRO [PREM C1 C2]
      `((EF PREM
            (LAMBDA SIMPLE [] ,C1)
            (LAMBDA SIMPLE [] ,C2)))))
```

However in our general attempt to support argument objectification even in intensional contexts, we will stay with the more complex definition in S5-184.

It is striking that EF must be primitive. If we associated Truth and Falsity with 0 and 1, respectively (if, in other words, we used the numerals 0 and 1 in place of the booleans $T and $F), then it would be trivial to define EF, as follows:

```
(DEFINE EF                                                    (S5-186)
   (LAMBDA SIMPLE [PREM C1 C2]              ; This is not a possible
      (NTH (+ 1 PREM) [C1 C2])))            ; definition of EF!
```

However such a suggestion would of course vitiate our aesthetic of category alignment. Given that we treat the booleans as a distinct structural class, we can see that the only other primitives that can take them as arguments (the only primitive functions, in other words, defined over truth-values) are SCONS, =, TYPE, NAME, and PREP (the last only in its first argument position). The essential task is to compare a truth-value or boolean, to select one or other of a pair of alternatives. Though = is of course capable of checking identity, and could therefore be used to compare the result of the premise against the booleans, it always yields another boolean, so that no comparison of a boolean with anything else would free us from the need to discharge a boolean. No solution, in other words, will emerge from a definition containing the following term:

```
... (= PREM $T) ...                                           (S5-187)
```

The only selector we have is NTH, which requires for its index a number; thus if it were possible to select one of *two* numbers based on having one of two booleans, a candidate definition could be found. However this task — choosing a *number* based on a truth-value — is essentially similar in structure to the original one: choosing a *consequent* based on a truth-value. In sum, though we do not offer formal proof, it should be clear that there is no way of composing these with other functions to yield EF behaviour non-primitively.

There is an alternative, suggested by S5-186: EF could be *replaced* with another primitive. In particular, if we defined a primitive procedure called BOOL to map Truth and Falsity, respectively, onto the numbers 0 and 1, we could then have the following definitions of both EF (IF could be defined in terms of EF as above, or could be given its own definition directly in terms of BOOL):

```
(DEFINE EF                                                    (S5-188)
   (LAMBDA SIMPLE [PREM C1 C2]
      (NTH (+ 1 (BOOL PREM)) [C1 C2])))
```

Very little rests on a choice between these two proposals (EF vs. BOOL as primitive); we will retain the extensional conditional.

Again we have a footnote. We pointed out above that the normalisation of the consequents of a conditional are iterative (tail-recursive) calls; it is also true that the reflected level calls to NORMALISE are tail-recursive as well (but tail-recursive at a different level). This last fact is crucial, as a discussion in section 5.c.iii will make clear.

5.b.viii. Review and Comparison with 2-LISP

It is instructive to review briefly the difficulties we encountered in our design of 2-LISP, and to show how the 3-LISP reflective capability has dealt with all of them. Six issues were of particular concern, as we mentioned at the end of chapter 4:

1. The relationship between environments and environment designators.

In 2-LISP environment designators crept into closures, but were not otherwise handled, and we left unsolved a rather major problem with our meta-theoretic characterisation: how these structural *environment designators* were to be kept synchronised with the environments posited in the meta-theoretic account. In 3-LISP the relationship between environments and environment designators is subsumed in the general issue of reflection: shifting levels is *guaranteed* to provide informationally correct designators of the context prior to the shift. Thus, although we do not have a mathematical account of reflection, we have made the correspondence between theory and structure explicit and well-behaved. The use of such designators *in object level closures* remains somewhat inelegant, but this level-crossing behaviour is not theoretically problematic.

2. The difficulty of using IMPRS in a statically scoped dialect.

We pointed out in section 4.d.iii that, partially in virtue of 2-LISP's static scoping, it was difficult to make good use of IMPRS, because the context of use of the non-normalised argument was not available to the body of the intensional procedure. Reflective procedures (REFLECTS) differ from intensional procedures (IMPRS) precisely in that they provide access not only to the (hyper-) intensional argument expression, but also to the context that was in

force at the point of use. Thus this difficulty is thoroughly dissolved.

3. Non-standard control operators.

We did not introduce any non-standard control operators into 2-LISP, but it was evident that if we had wanted any, they would have had to be provided primitively — there was no chance of constructing them in the language as defined. In contrast, we have already seen in 3-LISP that we can define such *outré* procedures as UNWIND-PROTECT, CATCH and THROW, QUIT, and so forth, each using just a few lines of code. Even more radical control structures could be defined using more subtle reflective procedures, as we will see later in the chapter. Again, this limitation of 2-LISP has been completely discharged.

4. The relationship between SET and REBIND.

In 2-LISP we had to provide SET primitively, and *also* had sometimes to use REBIND (such as when we wanted to modify an own variable of a procedure from the outside). Though it was clear that REBIND was more general, we could not define SET in its terms because we lacked the ability to provide the proper environment designator. 3-LISP's reflective capability of course overcomes this difficulty: SET was adequately defined in terms of REBIND in S5-130, above.

5. Different contexts for the two normalisations inherent in REFERENT.

Though we admitted it was less than elegant, in 2-LISP we were forced to execute the second normalisation mandated by REFERENT redexes in the context resulting from the first. In 3-LISP we were able conveniently to provide an additional argument to REFERENT enabling these contexts to be different. Furthermore, we were also able to define a function (CURRENT-ENVIRONMENT) that engendered the simpler 2-LISP behaviour for circumstances when that behaviour was appropriate.

6. The relationship between NORMALISE (the primitive processor) and MC-NORMALISE (the meta-circular processor).

In 2-LISP the primitive NORMALISE bore very little connection to the meta-circular MC-NORMALISE of section 4.d.vii. In 3-LISP, as the next section will make clear, the reflective processor, which subsumes the functions of the meta-circular processor, is also the primitive NORMALISE: it has the explicitness of the meta-circular processor with the causal grounding of the primitive processor. In addition, it provides abilities that *neither* NORMALISE nor MC-

NORMALISE did in 2-LISP: access to the state of a computation mid-stream. Thus the reflective processor unifies *three* capabilities of a standard LISP: mid-stream access to the state of a computation (something that is typically implementation-dependent), publicly-available names for the primitive processor function, and the support of explicit meta-circular code embodying a procedural theory of the computational significance of the processor function.

In addition, we can see even at this early stage in our investigation how 3-LISP has various properties that we predicted in chapter 1. First, it is clearly an inherently theory-relative dialect: the reflective protocols absolutely embody the "environment and continuation" theory of LISP in the very behaviour of the primitives. Second, it is simpler than 2-LISP in many ways: we were able, for example, to remove three of the 2-LISP primitives, defining them straightforwardly as simple procedures in a few lines of code.

In spite of this theoretical simplicity, we must not shrink from the fact that 3-LISP is infinite: everything we have said about the dialect this far implies that an infinite tower of processors, or at least processor states, must be (at least virtually) provided. The tractability of this infinite ascent remains as the main open question about the coherence of the formalism. It is a threat we will be able to defuse, but we must first investigate the structure of the reflective processor itself.

5.c. The Reflective Processor

Though simple examples of 3-LISP, like those in the previous section, can be understood on their own, it is difficult to understand 3-LISP reflective procedures in any depth, except with reference to the ***reflective processor***. We turn to this procedure in this section. Strictly, by "the processor" we refer to an active process; what makes it *reflective*, as suggested in section 5.a, is that it can be understood in terms of the processing of a particular program by what amounts to a type-identical copy of itself. Since we have only limited vocabulary for discussing processes *per se*, we will focus entirely on the procedures that the processor runs.

Superficially, the code for the reflective processor is similar to that of the meta-circular processors presented in previous chapters. There are four main functions of interest: NORMALISE, REDUCE, NORMALISE-RAIL, and READ-NORMALISE-PRINT (the others are subsidiary utilities, unchanged from chapter 4). NORMALISE and NORMALISE-RAIL are identical to their 2-LISP counterparts, except of course for the EXPR/SIMPLE conversion:

```
(DEFINE NORMALISE                                                    (S6-191)
   (LAMBDA SIMPLE [EXP ENV CONT]
      (COND [(NORMAL EXP) (CONT EXP)]
            [(ATOM EXP) (CONT (BINDING EXP ENV))]
            [(RAIL EXP) (NORMALISE-RAIL EXP ENV CONT)]
            [(PAIR EXP) (REDUCE (CAR EXP) (CDR EXP) ENV CONT)])))

(DEFINE NORMALISE-RAIL                                               (S6-192)
   (LAMBDA SIMPLE [RAIL ENV CONT]
      (IF (EMPTY RAIL)
          (CONT (RCONS))
          (NORMALISE (1ST RAIL) ENV
             (LAMBDA SIMPLE [ELEMENT!]
                (NORMALISE-RAIL (REST RAIL) ENV
                   (LAMBDA SIMPLE [REST!]
                      (CONT (PREP ELEMENT! REST!)))))))))
```

REDUCE will differ in certain respects; in this first version we ignore primitives and reflectives, and expand the call to EXPAND-CLOSURE, since we have only one instance of it (since the present dialect has primitives of only one procedural type):

```
(DEFINE REDUCE                                                       (S6-193)
   (LAMBDA SIMPLE [PROC ARGS ENV CONT]
      (NORMALISE PROC ENV
         (LAMBDA SIMPLE [PROC!]
            (SELECTQ (PROCEDURE-TYPE PROC!)
               [REFLECT ... ]
```

```
[SIMPLE (NORMALISE ARGS ENV
          (LAMBDA SIMPLE [ARGS!]
             (IF (PRIMITIVE PROC!)
                 ... deal with primitive simples ...
                 (NORMALISE (BODY PROC!)
                            (BIND (PATTERN PROC!)
                                  ARGS
                                  (ENV PROC!))
                            CONT))))]))))))
```

READ-NORMALISE-PRINT calls the procedure LEVEL (defined below) to determine the reflective level at which the code is being processed, giving the answer to PROMPT to print to the left of the caret, as the examples have shown. Otherwise it is unchanged:

```
(DEFINE READ-NORMALISE-PRINT                                    (S5-194)
   (LAMBDA SIMPLE [ENV]
      (BLOCK (PROMPT (LEVEL))
             (LET [[NORMAL-FORM (NORMALISE (READ) ENV ID)]]
                (BLOCK (PROMPT (LEVEL))
                       (PRINT NORMAL-FORM)
                       (READ-NORMALISE-PRINT ENV))))))
```

The most important difference between this and the 2-LISP version, of course, has to do with causal connection. Viewed as the code for a meta-circular processor, one would take these definitions in the following light: if they were processed by the primitive langauge processor, they should yield behaviour *equivalent* to that of the primitive processor, in the sense that they would compute the same function from expressions to expressions (i.e., NORMALISE would be provably equivalent to Ψ). From our reflective standpoint, however, we require something stronger: that from the point of view of any possible program, the behaviour of the primitive processor be *indistiguishable* from that engendered by this code, *even upon reflection.* In order to see what that comes to, we will consider import of the line left incomplete: the proper treatment of reflective redexes by REDUCE. It is in our treatment of that particular line where the substance of reflection will be manifested.

5.c.i. The Integration of Reflective Procedures

A reflective procedure is one that is run one level above simple functions, as if it were called as part of the processor itself. If a reflective redex is reduced, the first clause in the SELECTQ statement in the definition of REDUCE will be chosen. We want the reflective function to be called *directly*: not to be *mentioned by the processor code.* The latter would

suggest code of some form such as

```
(EXPAND-CLOSURE PROC! ARGS ... )                                    (S5-195)
```

just as in the case of the simple functions, but that would be to *process* a reflective procedure, from a level above: it would not include the function at the current level. Rather, we simply want to call it (not worrying, for the moment, about how it will itself get run). A first suggestion as to how to do this is the following:

```
(SELECTQ (PROCEDURE-TYPE PROC!)                                     (S5-196)
   [REFLECT (PROC! ARGS ENV CONT)]          ; This has a bug
   [SIMPLE ... ])
```

This has one correct property; it calls the reflective procedure with the proper three arguments: *designators* of the non-normalised arguments in the redex, of the environment in effect at the point of normalisation, and of the continuation with which the reduction was called. There is a problem, however, with PROC!: it is a variable used in the body of REDUCE as the name of a *function designator*, not a *function itself*. For example, if NORMALISE were called with the expression (CAR '(A . B)), REDUCE would be called with 'CAR and '['(A . B)] as arguments, PROC! would be bound to the designator of the binding of CAR in the appropriate environment: likely the primitive closure of the CAR function. But that closure is an *expression* (like all closures); therefore the redex we just wrote down — (PROC! ARGS ENV CONT) — is semantically ill-formed. We have made, in other words, a use/mention error; we intend instead to apply the function *designated by the referent of* PROC! to the arguments in question.

An apparently simple solution would be to dereference PROC! explicitly, as follows:

```
(SELECTQ (PROCEDURE-TYPE PROC!)                                     (S5-197)
   [REFLECT (↓PROC! ARGS ENV CONT)]       ; This has a different bug
   [SIMPLE ... ])
```

But this is a little too hasty: since PROC! designates a *reflective* closure (as the SELECTQ has just determined) ↓PROC! will *normalise* to that reflective closure. The consequent of the second line in S5-197, therefore, *is itself a reflective redex*, which will start up the processing of a line just like this one in the reflective processor that runs this one, and so on forever: it would engender an infinite number of reflections up the reflective hierarchy. This is not only infinite, it is wrong: we intended the reflective function to be run *at this level*, not to reflect again. We would like, in other words, to *apply the actual function designated by the*

referent of NORMAL-FUN, and functions are neither reflective nor simple. We do not want to *reduce a reflective procedure*; we want to reduce a *simple procedure that designates the function designated by the reflective procedure* PROC!.

The answer was suggested in section 5.a: if it were possible to define a different procedure PROC*, such that PROC* was a *simple* closure that designated the same function as PROC! and that had the same arguments and body as PROC, then that is the procedure we would like to *use* in the reflective processor. We will simply posit, therefore, a temporary function SIMPLIFY (not unlike the CORRESPONDING-FUN of S5-60) that converts REFLECTIVE closures to SIMPLE closures. Then our definition of REDUCE would look as follows:

```
(SELECTQ (PROCEDURE-TYPE PROC!)                                    (S5-198)
   [REFLECT (↓(SIMPLIFY PROC!) ARGS ENV CONT)]
   [SIMPLE ... ])
```

We can define an appropriate SIMPLIFY as follows:

```
(DEFINE SIMPLIFY                                                   (S5-199)
   (LAMBDA SIMPLE [REFLECTIVE-CLOSURE]
      ↑(SIMPLE . ↓(CDR REFLECTIVE-CLOSURE))))
```

These definitions, being purely structural, remain essentially unexplained: we need to inquire as to what function SIMPLIFY designates. An adequate answer, however, requires an answer to the prior question of what reflective closures designate in general: both topics are pursued in section 5.e, below. For the time being we will simply adopt the solution, dispense with the name SIMPLIFY, and insert the solution directly into the definition of REDUCE, omitting the redundant NAME and REFERENT operators. We arrive at the following definition:

```
(DEFINE REDUCE                                                     (S5-200)
   (LAMBDA SIMPLE [PROC ARGS ENV CONT]
      (NORMALISE PROC ENV
         (LAMBDA SIMPLE [PROC!]
            (SELECTQ (PROCEDURE-TYPE PROC!)
               [REFLECT ((SIMPLE . ↓(CDR PROC!)) ARGS ENV CONT)]
               [SIMPLE (NORMALISE ARGS ENV
                          (LAMBDA SIMPLE [ARGS!]
                             (IF (PRIMITIVE PROC!)
                                 ... deal with primitive simples ...
                                 (NORMALISE (BODY PROC!)
                                            (BIND (PATTERN PROC!)
                                                  ARGS
                                                  (ENV PROC!))
                                            CONT))))])))))
```

Aside from its omission of primitive functions, this definition will stand. It should be realised, however, that this simple introduction of ((SIMPLE . ↓(CDR PROC!)) ARGS ENV CONT) into the code has major and rather ramifying consequences. For one thing, it apparently renders the definition circular: these "meta-circular" programs were originally intended to explain how 3-LISP code is treated, and this last move has included in the midst of this supposedly *explanatory* program some of the code we were attempting to explain. We *used* what we were to have *mentioned* — a maneouvre that suggests vacuousness (although it must be admitted that even the "meta-circular" processors for 1-LISP and 2-LISP were formally vacuous as well, as betrayed by their names). Some explanation is due as to what S5-200 means — and, more particularly, how the resulting machine is finite. For with this one move we have already implied an infinite tower of processors. However we must first complete the processor definition.

5.c.ii. The Treatment of Primitives

We have not yet treated the primitives. In chapter 4 we used the following meta-circular definition of REDUCE-EXPR for 2-LISP:

```
(DEFINE REDUCE-EXPR                                                    (S5-201)
   (LAMBDA EXPR [PROCEDURE ARGS ENV CONT]
      (SELECT ↓PROCEDURE                              ; This is 2-LISP
         [REFERENT  (NORMALISE ↓(1ST ARGS) ENV CONT)]
         [NORMALISE (NORMALISE ↓(1ST ARGS) ENV
                       (LAMBDA SIMPLE [RESULT] (CONT ↑RESULT)))]
         [REDUCE    (REDUCE ↓(1ST ARGS) ↓(2ND ARGS) ENV
                       (LAMBDA SIMPLE [RESULT] (CONT ↑RESULT)))]
         [$T (CONT ↑(↓PROCEDURE . ↓ARGS))])))
```

It was important to deal explicitly with REFERENT, NORMALISE, and REDUCE, since they involved explicit normalisations beyond those implied by their being extensional procedures. This was not merely an aesthetic point: we had to make such processing explicit in order to ensure that the proper context arguments were used. In our present situation we must again manifest any explicit additional processing that is indicated by our primitives, for the same reason. Once again only these three primitives are candidates for special treatment: the rest will be adequately described, as they were in 2-LISP, by the term ↑(↓PROCEDURE . ↓ARGS).

However in 3-LISP we can reduce our concern about "special primitives" from three to one, for this reason: *we are in the midst of formulating definitions of* NORMALISE *and*

REDUCE: no *further* treatment needs to be given. It is for this reason that we said that NORMALISE and REDUCE were not like other 3-LISP primitives: we don't need to recognise them as special. If the user were merely to type in the definitions we are laying out, that would be perfectly adequate: they could be used as fully competent calls to the processor.

Why then do we say that NORMALISE and REDUCE are primitive at all? Because even though primitive closures of these two functions need not be recognised by the reflective processor, we are nevertheless defining the processor to be of such a form that, if it were to process the definitions we are spelling out, indistinguishable behaviour would result. In other words it is the *behaviour* of NORMALISE and REDUCE that is primitive, not the *designators* of that behaviour. If you formulate an importantly *different* definition of NORMALISE, it will be *wrong*: it will fail to designate the procedural function computed by the primitive processor. If, however, you formulate one that is *correct* (we will spell out a little more later about what "correct" comes to in this regard), then you can use that with impunity; no primitive binding needs to be used. In the initial 3-LISP environment, in other words, there are only twenty-seven primitive bindings, not twenty-nine.

Thus our explicit treatment of primitive simple closures needs to focus only on REFERENT. We said earlier that 3-LISP's REFERENT differed from 2-LISP's in that a second argument was used as a designator of the appropriate environment, rather than defaulting to the tacit context in present use. We said as well that the second normalisation was tail-recursive: it was given the same continuation as the original REFERENT redex. We are led, then, to the following characterisation:

```
(DEFINE REDUCE-EXPR                                              (S5-202)
   (LAMBDA SIMPLE [PROCEDURE ARGS ENV CONT]
      (IF (= PROCEDURE ↑REFERENT)
          (NORMALISE ↓(1ST ARGS) ↓(2ND ARGS) CONT)
          (CONT ↑(↓PROCEDURE . ↓ARGS)))))
```

Rather than have a specially-named procedure called REDUCE-EXPR (or even a 3-LISP version called REDUCE-SIMPLE), we can merely integrate this behaviour into the definition of REDUCE. For perspicuity we define a function called MAKE-C1 (we will explain that name later) that constructs an appropriate continuation for the recursive call when normalising SIMPLE argument expressions. In addition we re-arrange the tests to make things simpler:

```
(DEFINE REDUCE                                                   (S5-203)
   (LAMBDA SIMPLE [PROC ARGS ENV CONT]
      (NORMALISE PROC ENV
```

```
            (LAMBDA SIMPLE [PROC!]
               (SELECTQ (PROCEDURE-TYPE PROC!)
                  [REFLECT ((SIMPLE . ↓(CDR PROC!)) ARGS ENV CONT)]
                  [SIMPLE (NORMALISE ARGS ENV (MAKE-C1 PROC! CONT))])))))

(DEFINE MAKE-C1                                                    (S5-204)
   (LAMBDA SIMPLE [PROC! CONT]
      (LAMBDA SIMPLE [ARGS!]
         (COND [(= PROC! ↑REFERENT)
                (NORMALISE ↓(1ST ARGS!) ↓(2ND ARGS!) CONT)]
               [(PRIMITIVE PROC!) (CONT ↑(↓PROC! . ↓ARGS!))]
               [$T (NORMALISE (BODY PROC!)
                              (BIND (PATTERN PROC!) ARGS! (ENV PROC!))
                              CONT)])))))
```

This will stand as the official definition. Though simple, many of its consequences are yet to be explored.

We can see right away how fortunate we are in our ability to have no primitive reflectives. Suppose for example we had retained an intensional IF as a primitive procedure, with something like the following meta-circular characterisation (this is only able to treat rail CDRS — but it is better to remain simple here):

```
(DEFINE IF                                                         (S5-205)
   (LAMBDA REFLECT [[PREM C1 C2] ENV CONT]
      (NORMALISE PREM ENV
         (LAMBDA SIMPLE [PREM!]
            (IF (= PREM! '$T)
                (NORMALISE C1 ENV CONT)
                (NORMALISE C2 ENV CONT))))))
```

The reflective processor, upon encountering a conditional redex, would normalise the CAR and obtain a designator of a primitive IF reflective closure. It would not do to "SIMPLIFY" this in the second last line of S5-203, since that would construct a non-primitive closure, of roughly the form (<SIMPLE> E0 '[[PREM C1 C2] ENV CONT] '(NORMALISE ...)), to be processed by the reflective processor. This would again call NORMALISE, ultimately engendering vicious circularity (since IF appears in the body of this newly-constituted "simplified" closure). There would have to be a special check for primitive reflectives, just as there is a special check for primitive simples. We would be led approximately to the following:

```
(DEFINE REDUCE                                                     (S5-206)
   (LAMBDA SIMPLE [PROC ARGS ENV CONT]
      (NORMALISE PROC ENV
         (LAMBDA SIMPLE [PROC!]
            (SELECTQ (PROCEDURE-TYPE PROC!)
               [SIMPLE (NORMALISE ARGS ENV (MAKE-C1 CONT))]
```

```
[REFLECT
   (IF (= PROC! ↑IF)
       (LET [[[PREM C1 C2] ARGS]]
           (NORMALISE PREM ENV
              (LAMBDA SIMPLE [PREM!]
                 (IF (= PREM! '$T)
                     (NORMALISE C1 ENV CONT)
                     (NORMALISE C2 ENV CONT)));)
           ((SIMPLE . ↓(CDR PROC!)) ARGS ENV CONT)])))))
```

Though in one sense this is no less well-defined than anything else, it means that the *processor* must reflect in the course of processing object level code. Furthermore, since the reflected processor itself uses IF, this means that *every* one of the infinite number of processors must reflect in order to treat a single conditional at the object level. Strikingly, so long as we have no primitive reflectives this is not the case: the processor did not reflect in order to treat reflective code: *that was exactly the point of* SIMPLIFY.

We have, then, completed the reflective processor; a complete listing of the substantive part is given here (the attendant utilities can be derived from chapter 4 by making the SIMPLE/EXPR substitution):

The 3-LISP Reflective Processor (S6-207)

```
(DEFINE NORMALISE
   (LAMBDA SIMPLE [EXP ENV CONT]
      (COND [(NORMAL EXP) (CONT EXP)]
            [(ATOM EXP) (CONT (3INDING EXP ENV))]
            [(RAIL EXP) (NORMALISE-RAIL EXP ENV CONT)]
            [(PAIR EXP) (REDUCE (CAR EXP) (CDR EXP) ENV CONT)])))

(DEFINE REDUCE
   (LAMBDA SIMPLE [PROC ARGS ENV CONT]
      (NORMALISE PROC ENV
         (LAMBDA SIMPLE [PROC!]
            (SELECTQ (PROCEDURE-TYPE PROC!)
               [REFLECT ((SIMPLE . ↓(CDR PROC!)) ARGS ENV CONT)]
               [SIMPLE (NORMALISE ARGS ENV (MAKE-C1 PROC! CONT))]))))))

(DEFINE MAKE-C1
   (LAMBDA SIMPLE [PROC! CONT]
      (LAMBDA SIMPLE [ARGS!]
         (COND [(= PROC! ↑REFERENT)
                (NORMALISE ↓(1ST ARGS!) ↓(2ND ARGS!) CONT)]
               [(PRIMITIVE PROC!) (CONT ↑(↓PROC! . ↓ARGS!))]
               [$T (NORMALISE (BODY PROC!)
                              (BIND (PATTERN PROC!) ARGS! (ENV PROC!))
                              CONT)]))))

(DEFINE NORMALISE-RAIL
   (LAMBDA SIMPLE [RAIL ENV CONT]
      (IF (EMPTY RAIL)
          (CONT (RCONS))
          (NORMALISE (1ST RAIL) ENV
             (LAMBDA SIMPLE [ELEMENT!]
                (NORMALISE-RAIL (REST RAIL) ENV
                   (LAMBDA SIMPLE [REST!]
                      (CONT (PREP ELEMENT! REST!)))))))))

(DEFINE READ-NORMALISE-PRINT
   (LAMBDA SIMPLE [ENV]
      (BLOCK (PROMPT (LEVEL))
             (LET [[NORMAL-FORM (NORMALISE (READ) ENV ID)]]
                (BLOCK (PROMPT (LEVEL))
                       (PRINT NORMAL-FORM)
                       (READ-NORMALISE-PRINT ENV))))))
```

5.c.iii. Levels of READ-NORMALISE-PRINT

In a standard LISP, it is enough to say that EVAL is the main processor function, to show a simple definition of READ-EVAL-PRINT, and to claim that the top-level user interface is mediated by a call to this procedure. In 3-LISP, however, considerably more is required.

As we will explain in this section, it is not immediately obvious how the infinite set of 3-LISP processor levels is, so to speak, "initialised".

We gave a definition of 3-LISP's READ-NORMALISE-PRINT in S5-207 on the previous page. Suppose that we claimed only that a user interacted with this routine at "top level" (at reflective level 1), without offering any further explanation of how this came about. In addition, suppose that we were then to type the following expression to this reader:

```
1> ((LAMBDA REFLECT ? 'HELLO))                                    (S5-208)
```

It is of course clear that this is a reflective redex that will reflect and return the atom 'HELLO. What is also clear, given the definition of READ-NORMALISE-PRINT, is that the call to NORMALISE will be given that atom, which would be printed, and the cycle would repeat:

```
1> ((LAMBDA REFLECT ? 'HELLO))                                    (S5-209)
1> HELLO
1>
```

What is *not* clear, however, is who called READ-NORMALISE-PRINT. Thus, if we were instead to reflect *twice*, as in:

```
1> ((LAMBDA REFLECT ?                                             (S5-210)
       ((LAMBDA REFLECT ? 'BONJOUR))))
```

or in the equivalent:

```
1> ((LAMBDA REFLECT ? (RETURN 'BONJOUR)))                         (S5-211)
```

then all that we know is that the atom BONJOUR will be given to the caller of READ-NORMALISE-PRINT.

We can surmise (given the infinite number of reflective levels that we know are there) that READ-NORMALISE-PRINT was invoked in virtue of the normalisation of the redex

```
(READ-NORMALISE-PRINT <ENV0>)                                     (S5-212)
```

but there are various ways in which this could have come about. There are no such redexes in the reflective processor itself (except within the definition of READ-NORMALISE-PRINT itself, but that is no help), so if it occurs structurally it must occur in some other procedure. Furthermore, the problem recurses: though we do not yet know what invoked this redex, it is also reasonable to suppose that an analogous structure invoked that invocation, and so forth.

No unique answer is mandated by any of our prior concerns: this is rather an isolated problem, although it does demand a solution. Two general protocols seem suggested. One is that the normalisation of the READ-NORMALISE-PRINT redex was engendered by an explicit call to the processor, one level above it, of the form

```
(NORMALISE '(READ-NORMALISE-PRINT <ENV0>) <ENV1> <CONT1>)          (S6-213)
```

If we were to generalise this suggestion in the obvious way, we would expect that *this* redex would have been normalised in virtue of the level above it normalising the following expression:

```
(NORMALISE '(NORMALISE '(READ-NORMALISE-PRINT <ENV0>)              (S6-214)
                       <ENV1>
                       <CONT1>)
           <ENV2>
           <CONT2>)
```

And so on and so forth. There is no doubt that this schema could be extended indefinitely.

It would remain to specify the appropriate environment and continuation arguments. Regarding the first, we have already said that each level is provided with a level-specific "root" environment, consisting of the number of the level bound to the atom LEVEL, over the global environment. Thus we could fill in S6-214 as follows:

```
(NORMALISE '(NORMALISE '(READ-NORMALISE-PRINT GLOBAL)              (S6-215)
                       (PREP ['LEVEL '1] GLOBAL)
                       <CONT1>)
           (PREP ['LEVEL '2] GLOBAL)
           <CONT2>)
```

Again, this could be extended arbitrarily. However the continuation argument is more problematic. One obvious candidate would be the identity continuation, as follows (we continue to illustrate the level 3 expression, since it best manifests the essential structure):

```
(NORMALISE '(NORMALISE '(READ-NORMALISE-PRINT GLOBAL)              (S6-216)
                       (PREP ['LEVEL '1] GLOBAL)
                       ID)
           (PREP ['LEVEL '2] GLOBAL)
           ID)
```

However this proposal has an extremely odd and unacceptable consequence. Suppose that we took this as the correct initial structure (i.e., assumed that each level consisted of the appropriate version of this), and then normalised the expression given in S6-211. The BONJOUR would be lifted out of the READ-NORMALISE-PRINT, and handed to the identity

continuation at level 2. This would cause a handle to this atom to be handed to the identity continuation at level 3, and so on. At the end of time the top level of the hierarchy would be given an infinite degree handle to this atom, and the processor would (presumably) stop.

This seems extreme. It is for this reason that we have adopted a different strategy altogether. What we simply posit is this: at the beginning of time, the top level processor normalises the redex

```
(READ-NORMALISE-PRINT (PREP ['LEVEL '∞] GLOBAL))                    (S5-217)
```

This would cause the following to be printed at the process interface:

```
∞>                                                                  (S5-218)
```

We then posit further that (the lexicalisation of) approximately the same redex is given to READ as input:

```
∞>  (READ-NORMALISE-PRINT (PREP ['LEVEL '∞-1] GLOBAL))              (S5-219)
```

This would of course engender:

```
∞>  (READ-NORMALISE-PRINT (PREP ['LEVEL '∞-1] GLOBAL))              (S5-220)
∞-1>
```

And so on and so forth, until we get to the bottom:

```
∞>  (READ-NORMALISE-PRINT (PREP ['LEVEL '∞-1] GLOBAL))              (S5-221)
∞-1> (READ-NORMALISE-PRINT (PREP ['LEVEL '∞-2] GLOBAL))
∞-2> ...
     ...                ; An infinite number of intermediate steps
     ...
4> (READ-NORMALISE-PRINT (PREP ['LEVEL '3] GLOBAL))
3> (READ-NORMALISE-PRINT (PREP ['LEVEL '2] GLOBAL))
2> (READ-NORMALISE-PRINT (PREP ['LEVEL '1] GLOBAL))
1>
```

We presume that it is at this point — in this state — that the 3-LISP process is given to the user.

This scheme has the advantage, among other things, that any unsuspected return to a higher-level continuation that was not provided by the user will be printed at that level, rather than disturbing anything at any yet higher level. In addition, it is both general and simple, in that nothing *special* distinguishes the call to READ-NORMALISE-PRINT that the reader sees.

Because of the infinite number of calls, and because of the control structure of READ-NORMALISE-PRINT (which will be examined in more depth in the immediately next section), it is a consequence of this proposal that there is a continuation of one level of embedding at each reflective level, rather than the identity continuation (this is why otherwise untreated returns are adequately caught). Because of this fact, we have to make this proposal part of the definition of 3-LISP, since any finite implementation will have to simulate this infinite ascent of readers. However this protocol interacts only mildly with the rest of the 3-LISP definition; any number of other proposals could equally well have been chosen (such as the one suggested above, wherein an uncaught return would presumably engender an error). Nonetheless the present regimen will suit our purposes.

5.c.iv. Control Flow in the Reflective Processor

It is essential to lay bare the control flow through the processor code. The first thing to establish is that NORMALISE is intensionally *iterative*: that it is called tail-recursively. To show that it is true, we first present a copy of the listing given in S6-207, but annotated in ways we will shortly explain.

Control Dependencies in the Reflective Processor (S6-226)

```
(DEFINE NORMALISE
   (LAMBDA SIMPLE [EXP ENV CONT]
      (COND [(NORMAL EXP) (CONT EXP)]
            [(ATOM EXP) (CONT (BINDING ATOM ENV))]
            [(RAIL EXP) (NORMALISE-RAIL EXP ENV CONT)]
            [(PAIR EXP) (REDUCE (CAR EXP) (CDR EXP) ENV CONT)])))

(DEFINE REDUCE
   (LAMBDA SIMPLE [PROC ARGS ENV CONT]
      (NORMALISE PROC ENV
         (LAMBDA SIMPLE [PROC1]                          ; Continuation C0
            (SELECTQ (PROCEDURE-TYPE PROC1)
               [REFLECT ((SIMPLE . ↓(CDR PROC1)) ARGS ENV CONT)]
               [SIMPLE (NORMALISE ARGS ENV (MAKE-C1 PROC1 CONT))])))))

(DEFINE MAKE-C1
   (LAMBDA SIMPLE [PROC1 CONT]
      (LAMBDA SIMPLE [ARGS1]                             ; Continuation C1
         (COND [(= PROC1 ↑REFERENT)
                (NORMALISE ↓(1ST ARGS1) ↓(2ND ARGS1) CONT)]
               [(PRIMITIVE PROC1) (CONT ↑(↓PROC1 . ↓ARGS1))]
               [$T (NORMALISE (BODY PROC1)
                              (BIND (PATTERN PROC1) ARGS1 (ENV PROC1))
                              CONT)]))))

(DEFINE NORMALISE-RAIL
   (LAMBDA SIMPLE [RAIL ENV CONT]
      (IF (EMPTY RAIL)
          (CONT (RCONS))
          (NORMALISE (1ST RAIL) ENV
             (LAMBDA SIMPLE [ELEMENT1]                   ; Continuation C2
                (NORMALISE-RAIL (REST RAIL) ENV
                   (LAMBDA SIMPLE [REST1]                ; Continuation C3
                      (CONT (PREP ELEMENT1 REST1)))))))))

(DEFINE READ-NORMALISE-PRINT
   (LAMBDA SIMPLE [ENV]
      (BLOCK (PROMPT (LEVEL))
             (LET [[NORMAL-FORM (NORMALISE (READ) ENV ID)]]
                (BLOCK (PROMPT (LEVEL))
                       (PRINT NORMAL-FORM)
                       (READ-NORMALISE-PRINT ENV))))))
```

We can identify four classes of procedures that are called here:

1. Utilities (like RAIL and 1ST and SIMPLE) that in turn call only other utilities and primitives, thus engendering no recursion in the processor;
2. Main processor functions (like NORMALISE and NORMALISE-RAIL); and
3. *Continuations:* functions, designators of which are passed in each case to procedures that bind them to the parameter CONT.

We can ignore procedures of the first variety, since they do not contribute to the topology of the control paths. It is straightforward to analyse those of the second sort; with the help of the annotations in the preceding listing we will be able to analyse calls of the third sort as well.

In particular, there are nine composite function designators that together form the substance of the reflective processor: five named reflective processor functions of type 2 in the preceding list (NORMALISE, REDUCE, MAKE-C1, NORMALISE-RAIL, and READ-NORMALISE-PRINT), and four continuations (of type 3), generated within the named reflective processor functions, labelled C0 through C3, and notated in an italic face. We will call these nine closures the ***standard closures***, consisting of the five ***standard procedures*** and four ***standard continuations***. The continuations are passed to NORMALISE or NORMALISE-RAIL as a third argument; in each of those procedures that third argument is sometimes explicitly called. Those closures, being lexically scoped, will contain the closed bindings of a variety of processor variables. We will look at each of them in turn.

C0: The continuation constructed by REDUCE when it normalises the CAR of the redex (the function designator). It is closed in an environment in which PROC, ARGS, ENV, and CONT are bound to designators of the non-normalised function designator, the non-normalised argument designator, and the environment and continuation. Thus all C0 continuation designators will be of the following form:

```
(<SIMPLE> [['PROC ... ]['ARGS ... ]['ENV ... ]['CONT ... ] ... ]          (S5-226)
          '[PROC!]
          '(SELECTQ (PROCEDURE-TYPE PROC!) ... )))
```

It is evident from this example how the continuations embed: the closed environment embodied within the C0 continuation contains within it a binding of the variable CONT to the previous continuation.

C1: The continuation constructed by C0 in virtue of calling MAKE-C1 (we now see why we chose this name) when it normalises the arguments to a *simple* redex.

It is closed in an environment in which PROC! is bound to a designator of the normal-form simple function designator (i.e., to a handle of a simple closure), and in which CONT is bound to the original redex continuation. Thus C1 continuations will be of the following form:

```
(<SIMPLE> [['PROC! ... ]['CONT ... ] ... ]                    (S5-227)
          '[ARGS!]
          '(COND [(= PROC! ↑REFERENT) ... ))
```

C2: The continuation constructed by NORMALISE-RAIL when it normalises the first element of a rail (or rail fragment). It is closed in an environment in which RAIL is bound to the normal-form designator of the rail of non-normalised expressions of which the first is being normalised; it expects a single normal-form designator of that first element's referent. C2 continuations will be of the following form:

```
(<SIMPLE> [['RAIL ... ]['ENV ... ]['CONT ... ] ... ]          (S5-228)
          '[ELEMENT!]
          '(NORMALISE-RAIL (REST RAIL) ENV (LAMBDA ... )))
```

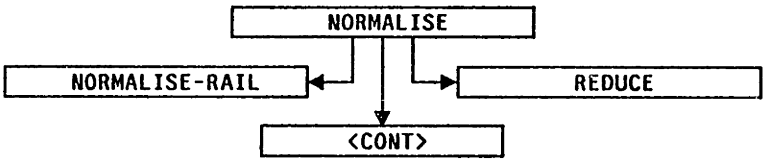
Note once again how the compositionality of the continuation structure is encoded in the embedded bindings of CONT.

C3: The continuation constructed by C2 when it normalises the remainder of a rail (or rail fragment) by calling NORMALISE-RAIL. It is closed in an environment in which ELEMENT! is bound to the normal-form designator of the element belonging ahead of the tail being normalised. In addition, since C3 continuations are always closed in C2 continuation bodies, the bindings in force for C2 continuations will also be in effect. C3 continuations will therefore be of the following form:

```
(<SIMPLE> [[ELEMENT! ... ]['RAIL ... ]['ENV ... ]['CONT ... ] ... ]   (S5-229)
          '[REST!]
          '(CONT (PREP ELEMENT! REST!)))
```

Given these identifications, we can begin to lay out the potential control flow for all possible paths through the reflective processor. We begin with NORMALISE; it is clear that it can call (tail-recursively in each case) any of three procedures: NORMALISE-RAIL, REDUCE, and CONT. Thus we approximately have the following beginnings of a control diagram:

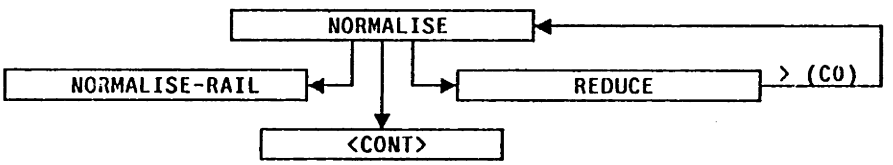
(S5-230)



We know what REDUCE and NORMALISE are; our goal is to *discharge* the CONT variable by tracing it around the entire loop. If we can figure out who calls NORMALISE and with what third argument, in other words, we can replace the un-informative "CONT" box in the above diagram with pointers to closures we can identify.

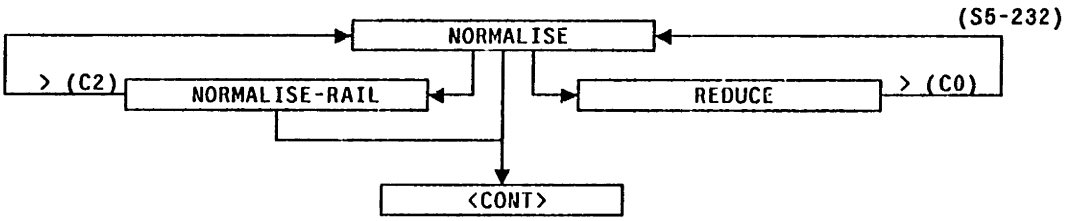
REDUCE *always* calls NORMALISE, with closure C0 as the continuation. Thus we add this line to our diagram, with the annotation "(C0)" on the line, indicating that this is the continuation argument (that will be bound to CONT in NORMALISE). In addition we have indicated with the sign ">" that C0 is an *embedding* continuation, in that it maintains within it the binding of the continuation that was passed in to REDUCE.

(S5-231)



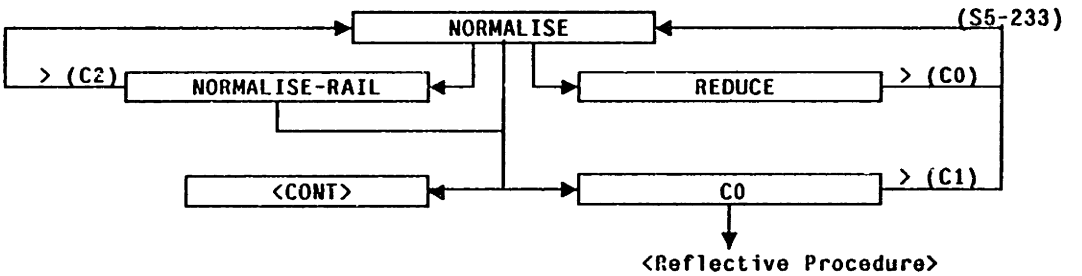
NORMALISE-RAIL can call either the continuation that it was passed, or else it can call NORMALISE with another embedding continuation C2, as indicated on the next version of our diagram. Furthermore, since NORMALISE-RAIL is called directly from NORMALISE, the continuation arguments that *it* can call are the same as those that NORMALISE can call directly; thus we indicate an arrow to the same "continuation" box yet to be discharged (NORMALISE-RAIL will also be called from C2, but again the same set of continuations will be involved):

(S5-232)

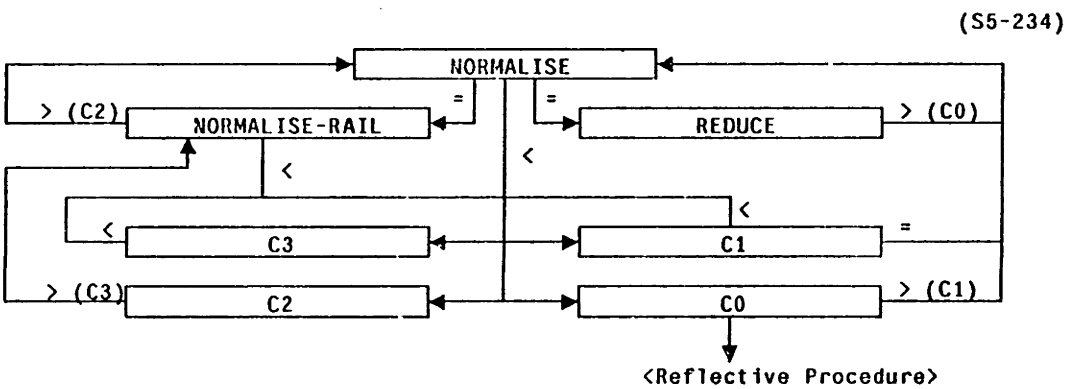


Now we can factor C0 out of the continuation box, since we know it is a possible continuation. C0 can call NORMALISE (if the redex is simple) with another embedding

continuation C1, or it can reflect, in which case all bets are off, because we have no way of knowing what the simplification of the user's reflective procedure will come to. These two alternatives are depicted in the following diagram:

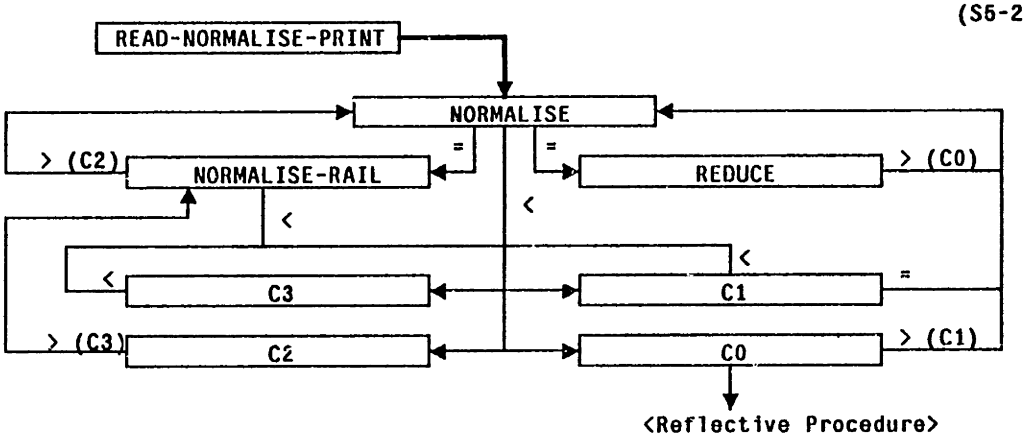


We still have C1, C2, and C3 to follow through. C1 takes one of three paths: either it calls NORMALISE with the continuation CONT that was embedded within it (on two of its paths), or else it calls that continuation directly. C2 *always* calls NORMALISE-RAIL, with an embedding continuation C3. Finally, C3 always calls its embedded continuation. We mark the direct calls to continuations (from NORMALISE and NORMALISE-RAIL as well as from C1 and C3) with "<", to indicate that the complexity of the passed continuations has *decreased*, rather than *increased* (on a standard implementation the stack would be popped, rather than grown). The diagram we now have looks as follows. This summarises all possible control flows except for intervening reflective procedures and READ-NORMALISE-PRINT:



Finally we look at the driver: READ-NORMALISE-PRINT. It calls NORMALISE with the identity continuation:

(S5-235)



Given this analysis, it is straightforward to establish that every call to a standard redex (every arrow in diagram S5-235) other than the call from READ-NORMALISE-PRINT to NORMALISE is tail recursive (intensionally iterative). In particular, in the listing presented in S5-225 we have italicised each of the continuation structures, to help distinguish them from the closures in which they appear. Furthermore, within each of the nine closures we have underlined the CAR of each redex that will be processed *with the same continuation the enclosing closure would be.* Consider for example NORMALISE: if a redex (NORMALISE A B C) were normalised with continuation C_k, then the COND redex would similarly be called with continuation C_k. Since COND (a macro) expands to a series of IF's, and since we know that the consequents of IFS are normalised iteratively, it is also true that the four consequents of each branch of NORMALISE's COND would also be called with continuation C_k. Thus in the listing the CAR of the COND redex and the CARS of the consequents of each of the COND consequents are underlined. Similarly in the other nine closures.

Consider then diagram S5-235 in conjunction with the listing in S5-225. NORMALISE calls NORMALISE-RAIL, REDUCE, and all continuations iteratively. Similarly, REDUCE calls NORMALISE iteratively. NORMALISE-RAIL calls either NORMALISE or its continuation argument iteratively. C0 calls NORMALISE iteratively; C1 similarly. And so on and so forth for C2 and C3. In fact the only *non-iterative call* in the processor (to one of the nine standard closures) is the call to NORMALISE within the body of the LET in READ-NORMALISE-PRINT — this is why it is indicated with a heavier line in S5-235.

This result should be understood in combination with the annotations (">", "<", and "=") on the arrows of S5-235. From the fact that all the arrows in the diagram represent

iterative tail-recursion, we can conclude that the entire state *of any given level of processing* will be reflected in the three arguments passed around: in the expression, environment, and continuation arguments passed around between NORMALISE, REDUCE, and so forth. We will use this fact in many places: in showing that 3-LISP is finite, in designing an acceptable implementation, and in encoding appropriate debugging protocols. What the three annotations on the arrows shows is how that explicitly passed state designator increases or decreases in complexity: each ">" implies that the continuation passed as an argument *embeds* the previous continuation (strictly, the previous continuation is bound to an atom in the environment over which the new continuation is closed); each "<" implies that such an embedded continuation is itself being called (implying that the continuation structure is decreasing in complexity); finally, an "=" signifies that the same continuation is passed from one standard closure to the next, in such a way that the continuation complexity is maintained.

These continuations, of course, may occur as the third argument to a reflective procedure — the one we always bind to the parameter CONT. This is notable because it is natural to ask, when writing a reflective procedure, about what possible arguments may be bound to the third parameter. There can be no *general* answer to this question, since higher level reflections may always call the processor with arbitrary functions as continuations. However a subsidiary question — and one to which we can provide a definite answer — is this: what will the binding of this variable be *if no previous reflective functions have altered them.* We will say that a reflective procedure is called with ***standard arguments*** if those arguments are of a form that could have arisen from the processing of arbitrary simple expressions at this level and lower, unaffected by the intervention of prior reflective code. Thus the third parameter, in a reflective procedure called with standard arguments, will be bound to a *standard continuation,* in exactly the sense that we defined that term earlier. Of the nine standard closures, in other words, only four can occur as standard continuations.

As our investigation of 3-LISP deepens it will turn out that a thorough understanding of the standard continuations will play a crucial role. For one thing, any implementation must be able to provide them as explicit arguments, *even if it has in fact run the prior code through some other mechanism than explicit processing through a copy of the reflective interpreter.* In the implementation presented in the appendix, for example, the

MACLISP routines run 3-LISP code directly, but at each step they put together the appropriate 3-LISP designators that *could be bound to* CONT (as well as to ENV and ARGS), in case some later reflective function accesses them. Though terribly inefficient, such functionality must be virtually provided, since the implementation cannot know when some reflective procedure may require access to the information they encode.

Note that there are an infinite number of possible distinct standard continuations: we have merely identified four *classes*, called C0 through C3. Nor are all instances of the C0 class even type-identical, for they can contain arbitrarily different bindings of the EXP argument, and arbitrarily different embedded continuations within them. What is important about C0 continuations is that *if* a reflective redex binds one to its CONT parameter, that implies that the redex occurred in the CAR of another redex. Similarly, if the CONT parameter is bound to a C1 continuation, the reflective redex occurred as the CDR of a simple redex. If the parameter is a C2 continuation, then the redex occurred as an element in a rail that was being normalised. Strikingly, there is no possibility of a redex being given a C3 continuation: only rails are normalised with such a continuation. However by looking at the embedded continuations bound within a given continuation, it is possible to encounter C3 continuations.

It is important finally to consider READ-NORMALISE-PRINT. It is a substantial design decision to have it *not* call NORMALISE iteratively; the opposite would always be possible, as the following code demonstrates:

```
(DEFINE READ-NORMALISE-PRINT                                    (S5-236)
  (LAMBDA SIMPLE [ENV]                      ; This is an alternative
    (BLOCK (PROMPT (LEVEL))                 ; definition of RNP.
           (NORMALISE (READ) ENV
              (LAMBDA SIMPLE [READ!]        ; This would be continuation C4
                 (BLOCK (PROMPT (LEVEL))
                        (PRINT READ!)
                        (READ-NORMALISE-PRINT ENV)))))))
```

We can see here why we had to use (RETURN ARGS) and (RETURN ENV) and (RETURN CONT) in our very first examples of reflective redexes, in S5-66 through S5-68. *If* we had adopted the definition just given in S5-236, instead of the actual version presented in S5-207, then a simple return at the reflected level would discard not only the continuation of the present computation, but would discard as well the entire continuation that was reading in and normalising and printing out expressions. In particular, proposed continuation C4 in S5-236

would be embedded in further continuations in the course of processing a given composite expression; continuation C4 contains the code that prints out the answer and reads in another expression. Thus by reflecting and ignoring the continuation one would dismiss the rest of the READ-NORMALISE-PRINT behaviour for this entire level, rather than simply shelving the continuation for this given expression.

However it does not seem reasonable that the continuation in a given continuation should include not only the state of the processor with respect to that computation, but also the potential for any further computation. In the popular wisdom a continuation is a function *from intermediate results to answers*; if we were to adopt the definition of READ-NORMALISE-PRINT given in S5-236 all continuations would be infinite (non-terminating) functions from intermediate results onto ⊥. It is for this reason that we have adopted the definition in S5-207. Under this regime, the continuation with which NORMALISE is called is the simple identity function; the rest of the READ-NORMALISE-PRINT function — the continuation that says that READ-NORMALISE-PRINT should loop — is embedded *in the continuation structure of the next level above.*

It should be noted in this regard that the identity function ID plays a very special role when used as a continuation: it seems to act as a function that flips the answer out from one tail-recursive program to the continuation of the caller up one level. Thus when an ID redex is encountered in the course of the reflective processor, the otherwise iterative NORMALISE ceases, and the result is handed to the continuation that initiated the cycle. However ID does not itself of course cross levels; this view of its role emerges only from the interaction between the tail-recursive NORMALISE and the other non-message-passing protocols we employ in programs that call NORMALISE.

It matters whether one embeds the processor or increases the complexity of continuations, as the following illustrative definitions of IF show (these are straightforward three argument versions that circularly use IF in the reflective processor — the sort of definitions that would be posited as generating the primitive closures, if IF were primitive). In the first NORMALISE is called tail-recursively, with the remaining structure of the computation embedded in the explicit continuation given as the third argument:

```
(DEFINE IF1 GLOBAL                                                    (S6-237)
   (LAMBDA REFLECT [[PREMISE C1 C2] ENV CONT]
      (NORMALISE PREMISE ENV                          ; This is an iterative
         (LAMBDA SIMPLE [PREMISE!]                    ; call to NORMALISE
            (IF (= RESULT 'ST)
                (NORMALISE C1 ENV CONT)
                (NORMALISE C2 ENV CONT))))))
```

In the second, we embed the continuation in variables bound in the reflective environment, and use as the continuation for the premise the simple identity function:

```
(DEFINE IF2 GLOBAL                                                    (S6-238)
   (LAMBDA REFLECT [[PREMISE C1 C2] ENV CONT]
      (LET [[PREMISE! (NORMALISE PREMISE ENV ID)]]  ; This is not
          (IF (= PREMISE! 'ST)
              (NORMALISE C1 ENV CONT)
              (NORMALISE C2 ENV CONT)))))
```

The difference would be manifested in reflective procedures that used or bypassed these continuations. For example, if we used the RETURN of S6-80 with the first, the returned value would be passed back over the conditional redex to the surrounding barrier:

```
1> (IF1 (= 1 (RETURN $F))                                             (S6-240)
        'YES
        'NO)
1> $F
```

In contrast, if we use IF_2 the answer would be returned only as the value of the premise:

```
1> (IF2 (= 1 (RETURN $F))                                             (S6-239)
        'YES
        'NO)
1> 'NO
```

Though there cannot be a final decision as to which is "right" and which "wrong", it seems unlikely that the first is intended: IF is not by and large thought of as a "barrier" at the reflective level, in the way that UNWIND-PROTECT and CATCH and the top level of READ-NORMALISE-PRINT are. We will therefore endorse the following general strategem:

> *Reflective code should call NORMALISE tail-recursively unless it has an explicit reason for not doing so (in which case it should be prepared to receive non-standard results, passing them through or otherwise treating them appropriately).*

In the examples we pursue in the next section we will honour this mandate by default, remarking explicitly in each case where we embed the reflective processor.

It should be noted in addition that the definition of IF we adopted — given in S6-184 — is in fact tail-recursive in this sense (it would therefore engender the behaviour

shown in S6-240). So too is the definition of SET given in S6-130. The only other reflective procedure that we have defined for standard use is LAMBDA: the version in S6-126 does not obey this mandate. It is, however, simple to define a version that does; the easiest approach is to define LAMBDA as we did before, as a first version, and then use it to define a properly iterative version, as follows (this is essentially a copy of the circular definition first introduced in S6-103):

```
(LET [[OLD-'AMBDA LAMBDA]]                                          (S6-241)
    (DEFINE LAMBDA
       (OLD-LAMBDA REFLECT [[TYPE PATTERN BODY] ENV CONT]
          (REDUCE TYPE ↑[ENV PATTERN BODY] ENV CONT))))
```

We will assume this redefinition in subsequent examples.

5.c.v. The Implementation of a Reflective Dialect

Given the analysis of reflective processor control flow in the previous section, we can see how a finite implementation of 3-LISP could be constructed. Our approach will be to review how a non-reflective dialect would typically be implemented, and then, with respect to such an implementation, to discuss what additional facilities would be required in the reflective case.

Nothing absolute can be said about implementation, of course, beyond the minimal satisfaction condition: all that is required is that the surface (behaviour) of the *implementing* process be interpretable, by an outside observer, as the surface of the *implemented* process, according to some conventional mapping. However there is a great deal of structure to the way in which implementations are *typically* built. In particular, one first establishes some encoding of the dialect's structural field in the structural field of the implementing language (a language we will generically call "IL" — it makes no difference what it actually is). Thus for example if we were to implement 2-LISP in a standard machine language, we might use pairs of adjacent memory cells to represent pairs, potentially longer sequences for rails, "pointers" to implement each of the first-order relationships (CAR, CDR, FIRST, REST, and PROPERTY-LIST), and so forth.

Once the protocols for encoding the field are fixed, one then constructs an IL program that, with respect to this encoding of structure, effects the behaviour of the dialect's Ψ. To continue with the 2-LISP example, we would construct an IL program that

took IL structures representing some 2-LISP structure as input, and produced as a result some other IL structure that represented the result of normalising the first. This IL program would itself be composite — recursive or iterative — according to the implementation design and the power of IL. In the course of normalising (encodings of) 2-LISP structures, this program would likely maintain state information in the form of environments and a stack (continuation structure). Other state information might be maintained, for example in tables to support input/output (an "oblist" to facilitate the correspondence between lexical items and their associated atoms, for example). There would in addition be utility routines to maintain the integrity of the mapping of the 2-LISP field into the IL field (memory management modules, garbage collectors, etc.).

Suppose that we had built a full implementation of 2-LISP along these lines, and that we then wanted to modify it to be an implementation of 3-LISP instead. The overarching mandate we have to satisfy is this: we will have to be able to provide, as full-fledged 3-LISP structures, designators of the environment and continuation information spelled out in the reflective processor. As implementors we of course have great freedom in our decision as to what constitutes an *implementation* of a full-fledged 3-LISP structure: we may want to put this information into the standard encoding we are already using (a simple but likely expensive approach), or we may want to leave it in a minimally complex form, and complicate our agreement as to what the mapping is between the two fields in question (a tricky but likely more efficient approach). For example, suppose that we have the continuation structure encoded in something like a stack in IL, and that we want to provide this information as a 3-LISP continuation designator (a closure). On the first approach we would build the (encoding of) the appropriate pair, presumably lifting the information from our stack and using it to form the closure as appropriate. On the second approach we would leave the information on the stack (or copy the stack fragment into some convenient place if necessary), and intercept all field accesses to see whether they pointed to this (type of) information. If so, CAR and CDR and so forth would be treated appropriately.

We mention all of these encoding concerns only to dispense with them: they can be handled by standard data encapsulation and data abstraction methods. We will simply assume that this is done somehow, and turn more crucially to the question of what information needs to be presented, and when.

If the 3-LISP program we were processing was entirely *simple* — included, that is to say, no reflective redexes — the 3-LISP implementation could (and probably should) proceed much in the way it did in the 2-LISP case. Suppose however that a reflective redex were encountered: we have to provide, for that redex, the appropriate three arguments: designators of the argument expression, the environment, and the continuation. The first is trivial; the second and third we can presumably construct in the manner just discussed. However what is crucial is that we have to *shift the level of the implementing process.* We have been assuming that the *implementing* processor has been running just one level above the explicit 3-LISP code — processing it directly, in other words, not in virtue of an intermediating level of reflective processor. When we encounter this reflective redex we have to shift back *into exactly the state we would have been in had we been running up one level from the very beginning.* In other words, suppose that we called the processor embodied by the implementation the *IL processor.* Then at any given point in the computation, the IL processor is simulating *one* of the processors in the infinite 3-LISP reflective hierarchy. By *shifting* the level of the IL processor we mean that we are *changing* which level of 3-LISP processor the IL processor is currently simulating. We must never think that 3-LISP *reflects: all* levels of the 3-LISP hierarchy of processors are always active, in the 3-LISP virtual machine.

It is at the point of shifting the level of the IL processor that the iterative nature of the 3-LISP processor is absolutely critical: it is relatively straightforward to figure out what environment and continuation structures would have been constructed had this deferred mode of processing been in effect since the beginning.

Suppose we call the environment and continuation structures actually used by the implementing processor the ***present context***. What is required, then, when the processor encounters a reflective redex, is that the present context be given to the reflective closure as arguments, and that a new present context be constructed, of exactly the form that it would have had, had the processor been running reflectively since the beginning. What we know, however, which is a great help, is that no non-standard reflective programs have previously been encountered: thus the appropriate new present context will simply consist of the single embedding mandated by READ-NORMALISE-PRINT.

Though it is comparatively straightforward to maintain the appropriate present context, it is not trivial. The best approach is to match the course of computation of the IL processor line by line with the 3-LISP reflective processor, and thereby determine exactly what information is required. For example, suppose that the IL processor is given a simple 3-LISP redex to normalise. If it had been running reflected, then this would be given to NORMALISE, which would bind it to the atom EXP, and would bind the current present context to the atoms EXP and CONT. Since there is no possibility of reflection within the body of NORMALISE, we will never need to reify this context, but we need nevertheless to know what the bindings are so that we can track subsequent calls.

Assuming that our simple redex is not a closure, the fourth clause of the COND redex in NORMALISE will be selected. We know, further, that REDUCE will be called with the CAR and CDR of the redex bound to PROC and ARGS, with the same present context bound once again to ENV and CONT. The CAR is next normalised, with a CO continuation as the continuation. We are about to recurse; what must be constructed somehow is an appropriate new present context containing enough information so that this CO continuation could be constructed (normalising the CAR, we must remember, might cause a reflection, in which case this CO continuation might have to be made available to some user code as an argument). In the simplisitic implementation presented in the appendix we actually *construct* the full (encoding) of the CO implementation, but this is far in excess of what is actually required: all we need to know is *that* it would be a CO continaution (two bits of information, since there are only four standard continuation types), and the bindings of PROC, ARGS, ENV, and CONT (four pointers).

And so on and so forth. When CONT arguments are called (such as in the first two clauses in NORMALISE) we can take the information we have retained and unpack it appropriately. Suppose for example that our simple redex is the structure "(+ X 3)"; then, we know that NORMALISE would take the second COND clause, resulting in the processing of (CONT (BINDING EXP ENV)). Our IL processor, therefore, will want to look up the binding of "+" in the environment, and then call CONT with the result. However we look at CONT and discover that it was a CO continuation; thus we know that we need to invoke that part of our IL processor that mimics the last four lines of REDUCE, with PROC and ARGS and the rest bound to what they were bound to when we constructed the CO continuation.

When we *do* encounter a reflective redex, we bind the pattern of the reflective closure to our reified present context, and adopt a new context as suggested above: the very simple one-level-deep context engendered by the top-level call to NORMALISE from READ-NORMALISE-PRINT.

This describes a theoretically viable implementation. We may call the obvious infinite implementation (a simple implementation running an infinite number of levels of reflective processor code) a ***class 0 implementation***, and the current proposal a ***class 1 implementation***. Unfortunately, although the class 1 implementation achieves mathematical finiteness, it has by no means yet achieved tractability. In particular, although the process that would result would behave correctly, it would be *extraordinarily* inefficient, for a reason we can readily see. In brief, our IL processor would be able to reflect *upwards*, but it would never *reflect down again*. Suppose for example that we had given it for processing the following structure:

```
((LAMBDA REFLECT [[] ARGS CONT] (CONT '3))                    (S6-242)
```

This is a reflective redex, which would cause the IL processor to shift upwards in the way we have talked about above. The issue we need to consider is what happens when this reflected processor processes the redex (CONT '3). CONT will be bound to a continuation structure that was reified by the IL processor just at the moment of reflection. Viewed from 3-LISP this continuation is a standard simple redex, of a form that was made clear in section 5.c.iv above. If the IL processor were to treat it this way — that simplest possibility — then from that moment on the IL processor would remain one reflected level above all subsequent simple 3-LISP structures. It was all very well to have "faked" this shift upwards, so as to look from the point of view of the reflected 3-LISP code as if we had always been stepped back, but it is equally crucial to come back down when this reflective posturing has lived out its usefulness.

This is not, we should make clear, a minor point of efficiency. The problem is made utterly serious by fact that the reflective processor contains reflective redexes (the SELECTQ at the beginning of all C0 continuations, for example); if the IL processor could only reflect upwards and could never reflect down, it would reflect upwards once again in running the C0 continuation. In other words once the first reflective redex in a 3-LISP program had been encountered, the IL processor would reflect upwards (with a concomitant loss of

efficiency) approximately half a dozen times *per subsequent operation.* The kind of inefficiency we are talking about, in other words, is devastating.

We must consider, then, what we will call a ***class 2 implementation.*** It is not difficult to have the IL processor put a special stamp on any structure handed to a reflective redex, so that if it encounters it at any future time as an argument to either NORMALISE or REDUCE, it can recognise that fact, and *reflect down again.* If those reified arguments are of the same form as those that the IL processor uses directly, then the shift down can be a very straightforward process; if they were explicitly constructed and in quite a different from the direct IL structures, then an appropriate backwards mapping will have to be performed. In any case this is all conceptually quite simple.

There is one subtlety to this downwards reflection, however: there is no guarantee about the complexity of the reflected processor when the downwards decision is made. Thus, as part of reflecting down, the present context of the IL processor must be *saved.* If and when a future reflective redex causes the IL processor to reflect upwards, that saved context must be used, rather than the very simple one from READ-NORMALISE-PRINT that we mentioned earlier. But again this is not difficult to implement: there must merely be a stack of contexts representing the state of each reflective processor *above* the one that the IL processor is currently simulating.

The overall idea is this: the IL processor operates as low in the reflective hierarchy as it can, at all times. When user-supplied reflective code is encountered, the IL processor no longer knows how to simulate the processor at its current level, so it has to climb one level higher, admitting the user-supplied code at the level it just vacated. However it keeps an eye on that level, and as soon as that user-supplied code *is no longer under direct scrutiny by its processor,* the IL processor knows that it can safely drop down again and resume its standard mimickry. It can do this not just when that user-supplied reflective code is *finished*; rather it does it *whenever it can,* even in the midst of such code.

This has been the briefest of sketches of what in full detail is a complex subject. Issues that we have not considered include the following:

1. How does one monitor the specially marked arguments given to reflective procedures to ensure that they have not been modified by the user? (Suppose someone does a RPLACA on a CO continuation, for example: can we be sure to note this?)

2. How should the implementation recognise explicit calls to the processor the form (NORMALISE <STRUCTURE> <ENV> <CONT>) or (REDUCE <P> <A> <ENV <CONT>)? Given that the NORMALISE definition need not be primitive, it is advantageous to recognise any type-equivalent definition, so that the IL processor can be used directly, rather than having to indirectly process the user-supplied definition. One approach would be to perform a quick check on every user-defined closure to see whether it is type-equivalent to the standard definition in the reflective processor (using hashing or some other efficient strategy). That the user uses the atom NORMALISE to refer to this closure is of course not something one wants to depend on: thus we should equally catch the embedded redex in:

```
(LET [[N NORMALISE]] (N <S> <E> <C>))                    (S6-243)
```

Alternatively, these two redexes *could* be made primitive, even though that is not strictly necessary (this is perhaps the most practical suggestion).

3. What is involved in supporting more than the minimal number of primitives in a 3-LISP implementation? Suppose for example that we wanted to make COND or SUBST an implementation primitive. What we must recognise is that if an argument to such a procedure were to reflect sufficiently, it could examine the continuation structure generated and determine, if the implementation is not very clever indeed, what is primitive and what is not (by seeing what expressions had been expanded and which had been treated in an indissoluable step). Thus it would seem that the only invisible way to add such primitives would be to force the IL processor to provide (presumably only virtually) the continuation and environments *that would have been constructed had the procedure been defined in the normal (non-primitive) manner.* This much at least is necessary if the extension is truly one of added efficiency, not changed behaviour.

These concerns may at first blush seem worrisome. And there are others perhaps even more major: what would it mean, for example, to *compile* a 3-LISP program? Certainly the general answers to all of these are beyond the scope of the present investigation, but the beginnings of an answer can be sketched.

The important insight is this: all of these concerns are *very similar one to another.* The point is that 3-LISP programs, being in a sense arbitrarily powerful (at least potentially), can wander around what must be virtually provided as an infinite hierarchy of explicit reflective levels. The only way that this can be implemented *at all* is for the implementing processor to mimic the lowest level of the infinite hierarchy such that, at that moment, every single level above it consists of exact copies of the primitive reflective

processor. Furthermore, this mimicking must be rather good: not only must the same *behaviour* ensue, but the same *trace* must be left, the same *structures* must be created, and so forth. Only if the mimicking is truly indistinguishable from all levels (except in terms of the passage of time, which we grant as an open parameter) can the implementation be called correct.

The issues raised in the last few pages have shown that the construction of such a correct implementation is not trivial. But the important thing to note is that no more information is kept by 3-LISP than in a standard dialect. In particular, the 3-LISP reflective processor does not automatically save records of all prior continuations or environments, which would increase the cost of an implementation categorically. Furthermore, since no more information is maintained than in a standard dialect, there is no reason that the *way* that it is kept in a standard dialect cannot suffice: the cleverness of implementation can be put into those routines that need to look at it, rather than into the processes that maintain it. In this way complex reflected procedures may be marginally slower than they might otherwise have been, but the standard and presumably overwhelmingly common behaviours will be engendered as speedily as they ever were.

In sum, while we do not deny that an implementation of 3-LISP may require some ingenuity, we see no reason why it needs to be inefficient.

5.d. Reflection in Practice

3-LISP is by now completely defined and explained. In this section we will present a number of examples that use its reflective powers, in part by way of illustration, and in part by way of suggestion. A few of the examples (like the one in section 5.d.ii on macros) are reasonably well worked out, but there are many issues raised here that should be investigated in much more depth: some of the examples merely point in the direction of interesting problems and inchoate solutions.

5.d.i. Continuations with a Variable Number of Arguments

We have seen that standard continuations are designed to accept a single argument; they are all of the form (LAMBDA SIMPLE [RESULT] ...). Because of the 3-LISP variable binding protocols, requiring the the pattern exactly match the argument structure, this means that exactly one argument must be supplied. However since continuations are regular procedures, it is of course possible to construct variants that demand no argument (literally, demand an empty sequence), or that demand several. We will explore a variety of such constructs in this section.

Consider first the following procedure, called NONE, which reflects and calls the continuation with no argument:

```
(DEFINE NONE                                                    (S5-249)
   (LAMBDA REFLECT [? ? CONT] (CONT)))
```

Any use of this function in a ***standard context*** — an extensional context normalised with a standard continuation — will cause an error, since too few arguments will have been supplied:

```
1> (NONE)                                                       (S5-250)
ERROR at level 2: Too few arguments supplied
1> (+ 2 (NONE))
ERROR at level 2: Too few arguments supplied
1> [1 (NONE) 3]
ERROR at level 2: Too few arguments supplied
```

The problem in each case is that the tacit continuation (the ID continuation supplied by READ-NORMALISE-PRINT in the first case, and a C3 continuation in each of the last two) required that a single expression be returned as the result of normalising a form, and

(NONE) called that continuation with none.

It is equally straightforward to define a procedure that returns several answers:

```
(DEFINE SEVERAL                                                    (S5-261)
   (LAMBDA REFLECT [? ? CONT] (CONT '2 '3 '4)))
```

Again, however, any use of this, given our current protocols, in a standard context will engender an error:

```
1> (SEVERAL)                                                       (S5-262)
ERROR at level 2: Too many arguments supplied
1> (+ 1 (SEVERAL))
ERROR at level 2: Too many arguments supplied
1> [1 (SEVERAL) 3]
ERROR at level 2: Too many arguments supplied
```

In order to use either NONE or SEVERAL, we would have to construct our own continuations to bind them. A simple example is this:

```
1> (NORMALISE '(SEVERAL) GLOBAL (LAMBDA SIMPLE ANSWERS ANSWERS))   (S5-263)
1> ['1 '2 '3]
```

We can also define a procedure called RECEIVE-MULTIPLE that explicitly accepts multiple replies from the normalisation of its (single) argument, and packages them together into a single sequence for *its* continuation:

```
(DFFINE RECEIVE-MULTIPLE                                           (S5-264)
   (LAMBDA REFLECT [[ARG] ENV CONT]
      (NORMALISE ARG ENV (LAMBDA SIMPLE ANSWERS (CONT ANSWERS)))))
```

We then have (note the use of the convention that a sequence of designators designates a sequence of their referents):

```
1> (RECEIVE-MULTIPLE (SEVERAL))                                    (S5-265)
1> [1 2 3]
```

Similarly, we can define a procedure that will happily normalise any expression, without demanding any reply at all:

```
1> (DEFINE RECEIVE-NONE                                            (S5-266)
      (LAMBDA REFLECT [[ARG] ENV CONT]
         (NORMALISE ARG ENV (LAMBDA SIMPLE ? (CONT ''OK)))))
1> RECEIVE-NONE
1> (RECEIVE-NONE (NONE))
1> 'OK
```

Though RECEIVE-NONE accepts no reply, it of course will not complain if a reply is given:

```
1> (RECEIVE-NONE 3)                                              (S5-257)
1> 'OK
1> (RECEIVE-NONE (PRINT 'HELLO)) HELLO
1> 'OK
1> (RECEIVE-NONE (SEVERAL))
1> 'OK
```

It is this last function — RECEIVE-NONE — that is of most interest, for it enables us to do what we promised to do in chapter 4: to define the side-effect primitives (RPLAC-, PRINT, and so forth) to return no answer. If we simply posit this change, then a simple use of PRINT in a standard context would cause an error:

```
1> (PRINT 'HELLO) HELLO                                          (S5-258)
ERROR at level 2: Too few arguments supplied
```

In a context where no answer is demanded, such as the argument position to RECEIVE-NONE, however, this revised PRINT will work acceptably:

```
1> (RECEIVE-NONE                                                 (S5-259)
     (PRINT '[IT IS A FAR FAR BETTER THING]))
   [IT IS A FAR FAR BETTER THING]
1> 'OK
```

In order to make this practice convenient, we would have to redefine BLOCK so as not to require answers from any except the last expression within its scope. The straightforward conversion of the 2-LISP definition of BLOCK that we have been assuming in 3-LISP is this (we call it BLOCK_1 to distinguish it from new definitions we will shortly introduce):

```
(DEFINE BLOCK1 (LAMBDA MACRO ARGS (BLOCK1* ARGS)))               (S5-260)

(DEFINE BLOCK1*                                                  (S5-261)
   (LAMBDA SIMPLE [ARGS]
      (COND [(EMPTY ARGS) (ERROR "Too few arguments supplied")]
            [(UNIT ARGS) (1ST ARGS)]
            `(LET [[? ,(1ST ARGS)]]
               ,(BLOCK1* (REST ARGS))))))
```

Our new definition, rather than being a MACRO, is a reflective function that does the sequential normalisation explicitly:

```
(DEFINE BLOCK2                                                   (S5-262)
   (LAMBDA REFLECT [ARGS ENV CONT] (BLOCK2* ARGS ENV CONT)))

(DEFINE BLOCK2*                                                  (S5-263)
   (LAMBDA SIMPLE [ARGS ENV CONT]
      (COND [(EMPTY ARGS) (CONT)]
            [(UNIT ARGS) (NORMALISE (1ST ARGS) ENV CONT)]
            [ST (NORMALISE (1ST ARGS) ENV
                  (LAMBDA SIMPLE ? (BLOCK2* (REST ARGS))))]])))
```

Note that $BLOCK_2$ does not *require* that the last expression within its scope return a result; rather, the normalisation of that last expression is simply given the BLOCK's continuation. Thus (BLOCK (PRINT 'HELLO)) will return no result, since the last expression within the BLOCK returned none. In addition, in distinction to $BLOCK_1$, $BLOCK_2$ need not have *any* expressions within its scope: in that case it returns no result on its own (no compelling behaviour suggested itself for a $BLOCK_1$ redex with no arguments, so we made that cause an error).

We can expect to use this just as we did before; the difference is that NONE and the newly re-defined side effect primitives will work correctly with it:

```
1> (BLOCK2 (TERPRI)                                              (S5-264)
           (PRINT 'YES-OR-NO?)
           (READ))
   YES-OR-NO? YES
1> 'YES
1> (BLOCK2 (NONE)
           (+ 2 3)
           (* 2 3))
1> 6
1> (BLOCK2 (NONE))
ERROR at level 2: Too few arguments supplied
1> (BLOCK2 (SET X '[COWBOYS AND INDIANS])
           (RPLACN 1 X 'COW-PERSONS)
           (RPLACN 3 X 'NATIVE-AMERICANS)
           X)
1> '[COW-PERSONS AND NATIVE-AMERICANS)
1> (RPLACT 2 X 'OR)
ERROR at level 2: Too few arguments supplied
```

These last examples illustrate an unfortunate side-effect of our new scheme: the top level driver (READ-NORMALISE-PRINT) is still a standard context, demanding a single reply. We could redefine READ-NORMALISE-PRINT to compensate for this, so that it will print out an answer only if one is returned (making it, in effect, a READ-NORMALISE-AND-MAYBE-PRINT):

```
1> (RPLACT 2 X 'OR)                                              (S5-265)
1>                                  ; No result is printed
```

Similarly, it should be possible to use multiple-value procedures at top level as well, as for example in:

```
1> (SEVERAL)                                                     (S5-266)
1> 2                                ; Three different results are printed
1> 3
1> 4
```

An appropriate re-definition of READ-NORMALISE-PRINT is the following:

```
(DEFINE READ-NORMALISE-PRINT                                          (S6-267)
   (LAMBDA SIMPLE [ENV]
      (BLOCK (PROMPT (LEVEL))
             (LET [[NORMAL-FORMS (NORMALISE (READ) ENV ID*)]]
                (BLOCK (MAP (LAMBDA SIMPLE [RESULT]
                               (BLOCK (PROMPT (LEVEL))
                                      (PRINT RESULT)))
                            NORMAL-FORMS)
                       (READ-NORMALISE-PRINT))))))
```

where ID* has the following definition:

```
(DEFINE ID* (LAMBDA SIMPLE ARGS ARGS))                                (S6-268)
```

Note that the innermost BLOCK in S6-267 will on this new scheme return no result, since its body ends with a PRINT redex. This would mean that MAP will be given no result, which on the present definition would cause an error, since MAP tries to return a sequence of the results of the element-by-element reductions. Any number of solutions are possible, which we needn't bother with here: a version of MAP could be defined that did not assemble results; the inner BLOCK could extended to return a dummy value; and so forth.

There is no need that BLOCK be a reflective procedure rather than a macro: the following version is identical in effect to that in S6-262.

```
(DEFINE BLOCK3                                                        (S6-260)
   (LAMBDA MACRO ARGS (BLOCK3* ARGS)))

(DEFINE BLOCK3*                                                       (S6-270)
   (LAMBDA SIMPLE [ARGS]
      (COND [(EMPTY ARGS) '(LAMBDA REFLECT [? ? CONT] (CONT))]
            [(UNIT ARGS) (1ST ARGS)]
            [$T `((LAMBDA REFLECT [? ENV CONT]
                     (NORMALISE ,(1ST ARGS) ENV
                        (LAMBDA SIMPLE ?
                           (NORMALISE ,(BLOCK3* (REST ARGS))
                                      ENV
                                      CONT)))))])))
```

This code works by wrapping all but the last expression inside a reflective application that normalises but ignores the result. Some examples will illustrate. In the following list, the first expression of each pair is expanded by the BLOCK3 macro into the second (thus we use the symbol "≡>", as in chapter 4, to indicate the first phase of macro reductions):

```
(BLOCK3 'HELLO)     ≡>    'HELLO                                      (S6-271)
```

```
(BLOCK3)            ≡>   (LAMBDA REFLECT [? ? CONT] (CONT))

(BLOCK3 (RPLACT 1 X '[NEW TAIL])
        (PRINT X)
        (1ST X))
   ≡>   ((LAMBDA REFLECT [? ENV CONT]
           (NORMALISE '(RPLACT 1 X '[NEW TAIL]) ENV
              (LAMBDA SIMPLE ?
                 (NORMALISE
                    '((LAMBDA REFLECT [? ENV CONT]
                        (NORMALISE '(PRINT X) ENV
                           (LAMBDA SIMPLE ?
                              (NORMALISE '(1ST X) ENV CONT)))))
                     ENV CONT)))))
```

Though this will work correctly, it is rather inelegant, in that it causes a reflective drop to a reflective application (i.e., a drop followed immediately by a jump back up) between each pair of expressions except the last. A seemingly better expansion for the second of these two pairs would be this:

```
((LAMBDA REFLECT [? ENV CONT]                                          (S6-272)
    (NORMALISE '(RPLACT 1 X '[NEW TAIL]) ENV
       (LAMBDA SIMPLE ?
          (NORMALISE '(PRINT X) ENV
             (LAMBDA SIMPLE ?
                (NORMALISE '(1ST X) ENV CONT)))))))
```

This reflects up just once, and then normalises each expression in turn, giving all but the last a special no-result continuation. The following definition of BLOCK will generate this code. Note that the role of the subsidiary BLOCK* has changed. A check for the single argument case is put into BLOCK4 itself, so that (BLOCK <EXP>) will expand into just <EXP>, rather than into ((LAMBDA REFLECT [? ENV CONT] (NORMALISE '<EXP> ENV CONT))), which is identical in effect but messy.

```
(DEFINE BLOCK4                                                         (S6-273)
   (LAMBDA MACRO ARGS
      (COND [(UNIT ARGS) (1ST ARGS)]
            [(EMPTY ARGS) '(LAMBDA REFLECT [? ? CONT] (CONT))]
            [$T `((LAMBDA REFLECT [? ENV CONT]
                    ,(BLOCK4* ARGS)))])))

(DEFINE BLOCK4*                                                        (S6-274)
   (LAMBDA SIMPLE [ARGS]
      `(NORMALISE ,(1ST ARGS) ENV
         ,(IF (UNIT ARGS)
              CONT
            `(LAMBDA SIMPLE ? ,(BLOCK4* (REST ARGS)))))))
```

We now have:

```
(BLOCK4 (+ 2 3))    ≡>   (+ 2 3)                                    (S6-276)

(BLOCK4)            ≡>   (LAMBDA REFLECT [? ? CONT] (CONT))

(BLOCK4 (PRINT 'YES-OR-NO?)
        (TERPRI)
        (READ))
   ≡>   ((LAMBDA REFLECT [? ENV CONT]
           (NORMALISE '(PRINT 'YES-OR-NO?) ENV
              (LAMBDA SIMPLE ?
                 (NORMALISE '(TERPRI) ENV
                    (LAMBDA SIMPLE ?
                       (NORMALISE '(READ) ENV CONT)))))))
```

Note that BLOCK4 is tail-recursive in the proper fashion: the final expression is normalised with the same continuation as was given to the whole BLOCK.

The importance of this exploration has been in showing how the reflective machinery has allowed us to use multiple results, and also no results, in a flexible way, without altering the underlying design of the dialect. We used the side effect primitives merely as illustrations of functions that arguably should not return a result: any other procedure might be given this status as well. We have not suggested that any of the primitives return *more* than a single result, although again a user procedure might avail itself of the possibility.

As a corollary to this main point, we are in a position to suggest that the 3-LISP side-effect primitives should return no result (and be given no declarative designation). In particular:

1. Redexes formed from the primitive procedures RPLACA, RPLACD, RPLACN, RPLACT, PRINT, and TERPRI would be defined to call the reflected level continuation with a null sequence. In terms of full procedural consequence their definitions will remain unaltered.
2. The definition of BLOCK4 would be accepted as the standard definition of BLOCK.
3. The definition of READ-NORMALISE-PRINT would be modified, perhaps as suggested in S6-267, so as to accept expressions that return either no or several results, as well as the default one.
4. The definitions of SET (S6-130) and DEFINE (S6-131) would, as a consequence of the former decisions, also return no result (although no redefinition is required).

It should be clear that these changes would not be made *in order to handle variable numbers of results.* Rather, we would adopt them *because* 3-LISP as presently defined is able to deal with variable numbers of results.

Two final remarks must be made. First, it would still be possible to define composite procedures that have side-effects *and* return results. For example, the following definition of RPLACT:NEW-TAIL would be just like the primitive RPLACT except that it would return the new tail (it is, in other words, exactly like 2-LISP's RPLACT):

```
(DEFINE RPLACT:NEW-TAIL GLOBAL                                    (S5-276)
   (LAMBDA SIMPLE [INDEX RAIL TAIL]
      (BLOCK (RPLACT INDEX RAIL TAIL) TAIL)))
```

Similarly, we could define a RPLACT:MODIFIED-RAIL, that returns as a result the entire modified rail:

```
(DEFINE RPLACT:MODIFIED-RAIL GLOBAL                               (S5-277)
   (LAMBDA SIMPLE [INDEX RAIL TAIL]
      (BLOCK (RPLACT INDEX RAIL TAIL) RAIL)))
```

Arbitrary other combinations are clearly possible.

Secondly, if we were to adopt this suggestion we would have to revamp the reflective processor slightly. As it stands, primitive procedures other than REFERENT are treated in a line of the following form:

```
(CONT ↑(↓PROCEDURE . ↓ARGS!))                                     (S5-278)
```

Expanded, this comes to:

```
(CONT (NAME ((REFERENT PROCEDURE (CURRENT-ENVIRONMENT))            (S5-279)
             . (REFERENT ARGS! (CURRENT-ENVIRONMENT))))
```

By our new conventions, if PROCEDURE designated one of the six primitive closures that return no results, then the NAME redex would cause an error, because the C2 continuation looking for its first argument would be given no answer. A revised MAKE-C1 of the following form could be used:

```
(DEFINE MAKE-C1                                                   (S5-280)
  (LAMBDA SIMPLE [PROC! CONT]
    (LAMBDA SIMPLE [ARGS!]
       (COND [(= PROC! ↑REFERENT)
              (NORMALISE ↓(1ST ARGS!) ↓(2ND ARGS!) CONT)]
             [(MEMBER PROC! ↑[RPLACN RPLACT RPLACA RPLACD PRINT TERPRI])
              (BLOCK (↓PROCEDURE . ↓ARGS!) (CONT))]
             [(PRIMITIVE PROC!) (CONT ↑(↓PROCEDURE . ↓ARGS!))])
```

```
        [$T (NORMALISE (BODY PROC!)
                       (BIND (PATTERN PROC!) ARGS! (ENV PROC!))
                       CONT)]))))
```

The reason that we are not yet in a position to accept this whole proposal, however, is that there are some decisions that would still need to be made. It is not clear, however, whether the strategy suggested in S5-280 is best: another would be to have C2 insert into the constructed sequence as many answers as were returned. Thus for example we would have:

```
[1 2 3]                              ⇒  [1 2 3]                  (S5-281)
[1 (NONE) 3]                         ⇒  [1 2]
[(SEVERAL) (NONE) (SEVERAL)]         ⇒  [1 2 3 1 2 3]
```

This has a certain elegance, although it also has a *major* consequence: the cardinality of the sequence designated by a rail would no longer be identifiable with the cardinality of the rail itself. The proposal would, however, allow the definition of MAKE-C1 in S5-207 to stand, and it would not require the complex definitions of BLOCK we have just constructed. On the other hand, it would perhaps be better to define a procedure, in the way we defined BLOCK, that would support this behaviour; thus we might have something like the following (assuming we chose the name COLLECT):

```
(COLLECT 1 2 3)                      ⇒  [1 2 3]                  (S5-281)
(COLLECT 1 (NONE) 3)                 ⇒  [1 2]
(COLLECT (SEVERAL) (NONE) (SEVERAL)) ⇒  [1 2 3 1 2 3]
```

This is area where further experimentation and thought is required, especially since there is no doubt that all of these various schemes are tractable. Furthermore, there are an entire range of related modifications to 3-LISP that should be considered if the dialect were to be used in a practical way, of which this is just one example. It would seem only reasonable, for example, to make the body expressions of LAMBDAS and LETS and COND clauses and so forth be implicit BLOCKS — this would allow the proposed definition of READ-NORMALISE-PRINT in S5-267 above to be more compactly written as follows:

```
(DEFINE READ-NORMALISE-PRINT                                    (S5-282)
   (LAMBDA SIMPLE [ENV]
      (PROMPT (LEVEL))
      (LET [[NORMAL-FORMS (NORMALISE (READ) ENV ID*)]]
          (MAP (LAMBDA SIMPLE [RESULT]
                  (PROMPT (LEVEL))
                  (PRINT RESULT))
               NORMAL-FORMS)
          (READ-NORMALISE-PRINT))))
```

There is in addition the question of whether it is reasonable to insist, as we have throughout, that a pattern match *all* arguments to a procedure. There would certainly be some convenience if extra arguments could be ignored, implying for example that ((LAMBDA SIMPLE [X] (+ X 1)) 3 5) would return 4. All of these suggestions have to do with sequences and cardinality, and should probably be considered together, so that a coherent policy would cover them all. Since it is not in our present interest to complicate the dialect, even in these ways that would simplify its surface use, we will defer the "returning no result" decision, with the assumption that it would be resolved before a practical system were constructed.

5.d.ii. Macros

We have used the procedure MACRO as a type argument to LAMBDA in prior examples, but we have said that MACROS are not primitive; therefore we still have to define the MACRO procedure. This is also a generally instructive exercise, in part because the proper treatment of macros provides an excellent example, in a nutshell, of many of the subtleties that characterise the proper use of procedural reflection. The issue is one of stopping the regular interpretation process in mid-stream, running a program to generate a structural representation of a procedure that is needed, and then dropping back down again to continue the interrupted computation using this new piece of code. The smooth integration of such a facility — and the ability to *define* such behaviour straightforwardly — are the kinds of characteristics we originally set out to provide in a reflective system.

The definition of LAMBDA in S5-103 shows how MACRO will be called: with three arguments: designators of an environment, a pattern, and a body. In 2-LISP all that was required was that this be turned into a MACRO closure, but because MACROS are not primitive some other type of closure — either SIMPLE or REFLECTIVE — will have to be constructed. It should be clear that it is a reflective closure that we will need.

The easiest way to see how MACRO should be defined is to show how they can be modelled using reflective functions. In particular, the following:

```
(DEFINE <NAME>                                                    (S5-283)
   (LAMBDA MACRO <PATTERN> .BODY>))
```

should be entirely equivalent in effect to:

```
(DEFINE <NAME>                                                    (S5-284)
   (LAMBDA REFLECT [ARGS ENV CONT]
      (NORMALISE (LET [[<PATTERN> ARGS]] <BODY>)
                 ENV CONT)))
```

For example (making use of the back-quote notation), suppose we defined the following:

```
(DEFINE INC                                                       (S5-285)
   (LAMBDA MACRO [X] `(+ 1 ,X)))
```

Then by our reconstruction this is to be equivalent to:

```
(DEFINE EQUIVALENT-TO-INC                                         (S5-286)
   (LAMBDA REFLECT [ARGS ENV CONT]
      (NORMALISE (LET [[[X] ARGS]] `(+ 1 ,X))
                 ENV CONT)))
```

This works as follows: Given a call to EQUIVALENT-TO-INC such as (EQUIVALENT-TO-INC (+ A B)), the processor will reflect and bind ARGS to '[(+ A B)], and ENV and CONT to the previously tacit environment and continuation structure as usual. Thus, using the fact that we can substitute bindings into expressions, the body of S5-286 would be equivalent in this case to:

```
(NORMALISE (LET [[[X] '[(+ A B)]]] `(+ 1 ,X))                     (S5-287)
           ENV CONT)))
```

Because of the automatic destructuring, X will be bound to '(+ A B), and `(+ 1 ,X) will normalise to '(+ 1 (+ A B)); therefore S5-287 will in turn be equivalent to:

```
(NORMALISE '(+ 1 (+ A B)) ENV CONT)                               (S5-288)
```

which is of course just right. What is perhaps most striking about this is the fact that no dereferencing of the code produced by the macro definition was required: its definition is merely a procedure that generates function designators, which should be run and then normalised, just as our example has shown. Since the macro is *run* at a reflective level, the fact that it generates a *function designator* rather than a real *function* is entirely appropriate: function designators are what NORMALISE and REDUCE need as explicit arguments.

We may turn then to the question of defining MACRO. We know that the reduction of the following generic LAMBDA redex:

```
(DEFINE <NAME>                                                    (S5-289)
   (LAMBDA MACRO <PATTERN> <BODY>))
```

will lead to the tail-recursive reduction of the following:

```
(MACRO <ENV> '<PATTERN> '<BODY>)                                    (S5-290)
```

Furthermore, we have just argued that this should generate the same structure as would result from the normalisation of the following alternative LAMBDA redex (we have expanded the LET of S5-284; we can't use LET to define MACRO because we will ultimately want to use MACRO to define LET):

```
(DEFINE <NAME>                                                      (S5-291)
   (LAMBDA REFLECT [ARGS ENV CONT]
      (NORMALISE ((LAMBDA SIMPLE <PATTERN> <BODY>) . ARGS)
                 ENV CONT)))
```

namely, to a redex of the following form:

```
(<REFLECT> <ENV>                                                    (S5-292)
           '[ARGS ENV CONT]
           '(NORMALISE ((LAMBDA SIMPLE <PATTERN> <BODY>) . ARGS)
                       ENV CONT)))
```

From these four facts we can readily define MACRO as follows:

```
(DEFINE MACRO                                                       (S5-293)
   (LAMBDA SIMPLE [DEF-ENV PATTERN BODY]
      (REFLECT DEF-ENV
               '[ARGS ENV CONT]
               `(NORMALISE ((LAMBDA SIMPLE ,PATTERN ,BODY) . ARGS)
                           ENV CONT)))
```

However this can be substantially simplified, by putting the pattern match directly in the reflective procedure's pattern:

```
(DEFINE MACRO                                                       (S5-294)
   (LAMBDA SIMPLE [DEF-ENV PATTERN BODY]
      (REFLECT DEF-ENV
               `[,PATTERN ENV CONT]
               `(NORMALISE ,BODY ENV CONT)))
```

DEF-ENV here is the "defining environment" — the environment in which the MACRO LAMBDA redex is closed; ENV is in contrast the environment in which the resulting closure is used. To see how these differ, we will look at some simple examples. Consider, for example, the following definition:

```
(DEFINE ADD-Y                                                       (S5-294)
   (LAMBDA MACRO [X] `(+ ,X Y))))
```

This is a macro that adds its argument *to whatever value* Y *has in the context where the macro is used.* For example, we would have:

```
(LET [[Y 100]]                                                    (S5-295)
   (ADD-Y (+ 1 2)))      ⇒    103
```

Quite different, however, is the following definition:

```
(DEFINE INCREMENT                                                 (S5-296)
   (LET [[Y 1]]
      (LAMBDA MACRO [X]
         `(+ ,X ,↑Y))))
```

This defines a macro that increments its argument, independent of the binding of Y in the environment in which the macro is reduced. The point is that the *macro* function is one reflective level removed from the simplification of the expression it generates, and the environment in which the *macro* function is to be run is the defining (lexical) environment — DEF-ENV in S5-293; the one in which the resultant expression is simplified is the one in effect where the macro is applied — ENV in S5-293. We would for example have:

```
(LET [[Y 23]] (ADD-Y Y))          ⇒    46                         (S5-297)
(LET [[Y 23]] (INCREMENT Y))      ⇒    24
```

We are all but done, but there is unfortunately one slight further problem: the familiar conflict between meta-structural argument decomposition, and non-rail CDRs. Consider for example our definition of INCREMENT in the following context:

```
(INCREMENT . (REST [2 3]))                                        (S5-298)
```

This will cause an error, because the macro assumes that it can decompose the argument position in a single element rail (in virtue of having its variable list be [X]). We could define a more general INCREMENT_2 as follows:

```
(DEFINE INCREMENT₂                                                (S5-299)
   (LAMBDA MACRO ARGS `(+ (1ST ,ARGS) 1)))
```

This will work, but it seems inelegant. For one thing, the problem is not unique to INCREMENT: *every* macro would seem to potentially suffer this problem, which would seem to imply that *no* macro definition should ever count on being able to destructure its arguments. This is, unfortunately, true for macros that do not necessarily plan on simplifying their arguments. One answer is afforded by the following insight: INCREMENT expands into a form that will simplify the argument positions: we are simply not

particularly interested, in this case, in whether the first argument position can be destructured *before* simplification, since we intend that position to be simplified in the case of its use. Furthermore (another example of the usefulness of a simplifier), we don't care if more than one simplification is engendered. This suggests that we define a new macro definition function called S-MACRO, that simplifies the argument positions in the macro call first, and then runs the macro definition over the resultant simplified expression. The obvious first definition of S-MACRO is this:

```
(DEFINE S-MACRO                                                    (S5-300)
   (LAMBDA SIMPLE [DEF-ENV VARS BODY]
      (REFLECT DEF-ENV
               '[ARGS ENV CONT]
               `(NORMALISE ARGS ENV
                    (LAMBDA SIMPLE [ARGS!]
                       (NORMALISE ((LAMBDA SIMPLE ,PATTERN ,BODY)
                                   . ARGS!)
                                  ENV CONT))))))
```

However this is redundant; we can use the PATTERN argument directly in the continuation. Thus the following is equivalent but simpler:

```
(DEFINE S-MACRO                                                    (S5-301)
   (LAMBDA SIMPLE [DEF-ENV VARS BODY]
      (REFLECT DEF-ENV
               '[ARGS ENV CONT]
               `(NORMALISE ARGS ENV
                    (LAMBDA SIMPLE [,PATTERN]
                       (NORMALISE ,BODY ENV CONT))))))
```

If we now define INC in terms of this new function:

```
(DEFINE INC                                                        (S5-302)
   (LAMBDA S-MACRO [X] `(+ ,X 1)))
```

we will facilitate such uses as the following:

```
(INC 3)                          ⇒   4                             (S5-303)
(INC . (TAIL 3 [2 4 6 8]))       ⇒   9
(MAP INC [1 2 3 4])              ⇒   [2 3 4 5]
```

One final comment is warranted: not a problem, but an illustration of the elegance of our solution. In standard LISP's, it is possible to construct macro definitions that, while legal, are generally considered to be counter to the proper "spirit" of macros. In addition they cannot be compiled (although that should be taken as symptomatic of a problem, not in *itself* cause for rejection). An example from MACLISP is the following:

```
(DEFUN UNFORTUNATE MACRO (X)                                    (S5-304)
   (COND ((LESS (EVAL X) 0) `(+ ,X 1))     ; This is MACLISP
         (T `(- ,X 1))))
```

The problem is that the *definition* of the macro makes reference to the run-time value of the variable X. Nothing in the normal definitions of LISP or macros, however, actually excludes such definitions, and they will indeed work (interpreted). Somehow, though, one is not supposed to do this.

Suppose we try to construct the 3-LISP version of this macro:

```
(DEFINE UNFORTUNATE                                             (S5-305)
    (LAMBDA MACRO [X]
       (IF (LESS ↓(NORMALISE X <ENV> ID) 0)
           `(+ ,X 1)
           `(- ,X 1))))
```

This will simply fail, pretty much independent of what we use for the <ENV> argument to NORMALISE. The NORMALISE redex occurs in a lexically scoped function that is closed in the *defining* environment (DEF-ENV of the example above). Even if we were able to obtain an access to this environment, as for example in the following expression, the referent of X will surely not be bound *there.*

```
(DEFINE UNFORTUNATE                                             (S5-306)
    (LAMBDA MACRO [X]
       (IF (LESS ↓(NORMALISE X (CURRENT-ENVIRONMENT) ID) 0)
           `(+ ,X 1)
           `(- ,X 1))))
```

This call to CURRENT-ENVIRONMENT will obtain access to DEF-ENV extended with bindings of ARGS, ENV, and CONT, but X will not (in general) be bound in this. Presumably the author of this definition intended the simplification of X to be carried out *in the environment of the macro redex itself.* But — and this is the crucial point — S5-293 makes it clear that the function that *generates the program structure* is run at a reflected level, and in the defining environment. Thus our 3-LISP reconstruction not only shows why S5-304 merits its name, but it in addition prevents such functionality from being defined. Once again, tacit intuitions about programming practices turn out to be explicitly reconstructed in the 3-LISP framework.

This development has been intended to illustrate a variety of points. First, in dealing with macros clear thinking about environments and levels of designation is crucial

to success. Second, we have shown how reflection can be used to extend the apparent power of an interpreter: by adding a function that causes the interpretation process to reflect and construct a new form to normalise, and then to drop back down again with this new form in hand — this is one of the prime *desiderata* on reflective thinking we mentioned in the introduction and in the prologue. Macro definitions, it turns out, provide a good example of such reflective manipulation in a computational setting. Finally, as the case of S-MACRO illustrated, circumstances often arise where a behaviour slightly different from the most general case proves convenient; the fact that we could *define* MACRO made it easy to also define a minor variant.

5.d.iii. Pointers to Further Examples

It has been our task in this dissertation to develop 3-LISP; a full exploration of its powers, and of the practical uses of reflection, remains a topic for future research. In this last sub-section we will simply sketch very briefly a number of issues where the use of reflection seems indicated.

First, at various points the the foregoing discussion we have talked of dynamically scoped variables, of the sort that were supported primitively in 1-LISP. It should be clear that the UNWIND-PROTECT of S6-85 is sufficiently powerful to define a dynamically-scoped variable protocol: at each point where a dynamically scoped variable was to be bound, the code would reflect, save the prior binding, and *set* the (global) binding of the variable in question to its dynamic value. For example, we have suggested that we might support the following kind of structures (since pairs in patterns are otherwise unused):

```
(LET [[(DYNAMIC ERROR) 0.1]]                                    (S6-310)
     (SQUARE-ROOT 2))
```

It is then assume that the use of the redex (DYNAMIC ERROR) in a standard context, within, say, the body of the SQUARE-ROOT procedure, would provide access to the binding established in S6-310

Our current point is that this functionality could be implemented by expanding the foregoing kind of structure into:

```
(UNWIND-PROTECT                                                 (S6-311)
     (BLOCK (PUSH ERROR ERROR-SAVE)
            (SET ERROR 0.1)
            (SQUARE-ROOT 2))
     (POP ERROR ERROR-SAVE))
```

However this is far from an elegant solution. What it does is to establish an environment protocol that tracks the continuation structure, rather than one that follows the normal environment scoping rules; if this is the intent, a more perspicuous proposal would be to have a dynamic environment explicitly maintained by the processor, passed around explicitly, objectifiable upon reflection, and so forth (approximately this suggestion is presented in Steele and Sussman[1]). An appropriately modified processor would look approximately as follows (the new or modified parts are underlined):

```
(DEFINE NORMALISE                                                    (S5-312)
   (LAMBDA SIMPLE [EXP ENV DYN CONT]
      (COND [(NORMAL EXP) (CONT EXP)]
            [(ATOM EXP) (CONT (BINDING EXP ENV))]
            [(RAIL EXP) (NORMALISE-RAIL EXP ENV DYN CONT)]
            [(PAIR EXP) (REDUCE (CAR EXP) (CDR EXP) ENV DYN CONT)])))

(DEFINE REDUCE                                                       (S5-313)
  (LAMBDA SIMPLE [PROC ARGS ENV DYN CONT]
     (NORMALISE PROC ENV
        (LAMBDA SIMPLE [PROC!]
           (SELECTQ (PROCEDURE-TYPE PROC!)
             [REFLECT ((SIMPLE . ↓(CDR PROC!)) ARGS ENV DYN CONT)]
             [SIMPLE (NORMALISE ARGS ENV DYN (MAKE-C1 PROC! DYN CONT))]))))))

(DEFINE MAKE-C1                                                      (S5-314)
   (LAMBDA SIMPLE [PROC! DYN CONT]
      (LAMBDA SIMPLE [ARGS!]
         (COND [(= PROC! ↑REFERENT)
                (NORMALISE ↓(1ST ARGS!) ↓(2ND ARGS) DYN CONT)]
               [(PRIMITIVE PROC!) (CONT ↑(↓PROC! . ↓ARGS!))]
               [$T (LET [[[NEW-ENV NEW-DYN]
                          (BIND (PATTERN PROC!) ARGS! (ENV PROC!) DYN)]]
                     (NORMALISE (BODY PROC!) NEW-ENV NEW-DYN CONT))]))))

(DEFINE NORMALISE-RAIL                                               (S5-315)
  (LAMBDA SIMPLE [RAIL ENV DYN CONT]
     (IF (EMPTY RAIL)
         (CONT (RCONS))
         (NORMALISE (1ST RAIL) ENV DYN
            (LAMBDA SIMPLE [ELEMENT!]
               (NORMALISE-RAIL (REST RAIL) ENV DYN
                  (LAMBDA SIMPLE [REST!] (CONT (PREP ELEMENT! REST!)))))))))

(DEFINE READ-NORMALISE-PRINT                                         (S5-316)
   (LAMBDA SIMPLE [ENV DYN]
      (BLOCK (PROMPT (LEVEL))
             (LET [[NORMAL-FORM (NORMALISE (READ) ENV DYN ID)]]
                (BLOCK (PROMPT (LEVEL))
                       (PRINT NORMAL-FORM)
                       (READ-NORMALISE-PRINT ENV DYN))))))

(DEFINE BIND                                                         (S5-317)
   (LAMBDA SIMPLE [PATTERN ARGS ENV DYN]
      (LET [[[E-BINDINGS D-BINDINGS] (MATCH PATTERN ARGS)]]
         [(JOIN E-BINDINGS ENV) (JOIN D-BINDINGS DYN)])))

(DEFINE MATCH                                                        (S5-318)
   (LAMBDA SIMPLE [PATTERN ARGS]
      (COND [(ATOM PATTERN) [[[PATTERN ARGS]] (SCONS)]]
            [(AND (PAIR PATTERN) (= (CAR PATTERN) 'DYNAMIC))
             [(SCONS) [[PATTERN ARGS]]]]
            [(HANDLE ARGS) (MATCH PATTERN (MAP NAME !ARGS))]
            [(AND (EMPTY PATTERN) (EMPTY ARGS)) [(SCONS) (SCONS)]]
            [(EMPTY PATTERN) (ERROR "Too many arguments supplied")]
            [(EMPTY ARGS) (ERROR "Too few arguments supplied")]
            [$T (LET [[[E1S D1S] (MATCH (1ST PATTERN) (1ST ARGS))]
```

```
                [[E2S D2S] (MATCH (REST PATTERN) (REST ARGS) ,1]
            [(JOIN E1S E2S) (JOIN D1S D2S)]]])))
```

This code may seem odd in part because the dynamic environment is never used. However, given this protocol, we can *define* the procedure called DYNAMIC to support the behaviour suggested earlier, as follows:

```
(DEFINE DYNAMIC                                                    (S5-319)
   (LAMBDA REFLECT [[VAR] ENV DYN CONT]
      (CONT (BINDING VAR DYN))))
```

The similarity to the treatment of atoms in NORMALISE is evident.

If we were to adopt dynamically scoped free variables as a primitive part of 3-LISP, the reflective processor would look approximately as above (except perhaps a more efficient MATCH algorithm would be adopted). However it is important to realise that the code just given can be used explicitly, even in the dialect as current defined. In particular, it would be possible, if dynamically scoped free variables were required, to reflect at any point and to *use* the processor just presented. From an implementation standpoint the code that was run during this processing would necessarily be treated less efficiently than normally, but from a theoretical point of view no problems would arise. It must be admitted, however, that this would provide dynamic scoping *only at a given reflective level*, since reflective redexes processed by this processor (assuming that this processor was itself processed by the standard primitive 3-LISP processor) would be processed not by this NORMALISE, but by the standard 3-LISP processor.

On obvious question to ask is what would be required in providing a new definition of NORMALISE that would take effect at *all* reflective levels, but we will not answer that here; it would lead us into a much larger subject that this dissertation does not attempt to treat.

There are other reflective procedures that should be examined. We have not, for example, presented routines that examine and unpack continuation structures: although the basic structure of continuations was explained in section 5.c.iv, convenient debugging and error routines built on top of these remain to be developed. Similarly, we could explore modifications to the processor to support the tracing and advising of arbitrary procedures. Again, we might want to explore user interrupts, which would presumably cause the processor to reflect not because it encountered a reflective redex, but because of an external event. One can imagine, in other words, a modified processor of approximately the

following form:

```
(DEFINE NORMALISE                                                    (S6-320)
   (LAMBDA SIMPLE [EXP ENV CONT]
      (COND [(PENDING-INTERRUPT) ((GET-INTERRUPT-ROUTINE) EXP ENV CONT)]
            [(NORMAL EXP) (CONT EXP)]
            [(ATOM EXP) (CONT (BINDING EXP ENV))]
            [(RAIL EXP) (NORMALISE-RAIL EXP ENV CONT)]
            [(PAIR EXP) (REDUCE (CAR EXP) (CDR EXP) ENV CONT)])))
```

The interrupt routine, if it wished to resume the computation, would merely need to normalise (NORMALISE EXP ENV CONT); if it wished to abort it, it could simply return.

As was the case with respect to dynamic environments, this sort of modified definition of NORMALISE could either be made part of the primitive reflective processor (i.e., the dialect could be altered), or, more interestingly, such a processor could be run, indirectly, during any part of any other process. It is important to realise that the fundamental ability to reflect allows the *integration* of these modified processors into the normal course of a computation, with at worst a cost in efficiency. Such an ability is not possible in any prior dialect.

Two other areas of exploration are the simulation of multi-processing schemes, and more complex non-standard control protocols. It is striking that the definitions of the non-standard control primitives that have been provided in standard LISPS (THROW, CATCH, UNWIND-PROTECT, and so forth) are definable in 3-LISP in just one or two lines. There is no reason to suppose that, once provided with a reflective capability, much more complex or more subtle forms of reflective control might not be found useful. Again, we emphasise that it is not a contribution of the present research to suggest such regimes; our point is merely that 3-LISP can provide an appropriate environment in which such explorations could be easily conducted.

5.e. The Mathematical Characterisation of Reflection

Another open research problem emerging from our analysis has to do with the appropriate mathematical characterisation of procedural reflection in general, and of 3-LISP in particular. At present we do not even have any suggestions as to how such an account might be formulated, except to reject out of hand any attempt to construct the mathematical analogue of the implementation presented in the appendix (even though that would presumably lead in some formal sense to a tractable description). The problems stem from the infinite tower of processors; what we would like is a mathematical technique that would construct the appropriate limit of this recursive ascent, in much the way that the fixed point theorem establishes the limit of another kind of infinite recursive description.

In spite of a certain superficial similarity between recursion and reflection, it is important to recognise that there is a fundamental difference between the kind of infinite "self-reference" involved in recursive definitions, and the kind implicated in the tower of reflective processors. As we mentioned in section 4.c.v, the former remains always at the same level, whereas the latter involves a use/mention shift at each stage. It is for this reason that, at least so far as this author can presently see, no simple reconfiguring of the problem of reflection will put it into a form in which standard fixed-point results will apply. Rather, it seems likely that some abstract characterisation of the "finite amount of information" embodied in the 3-LISP processor would have to be developed, as well perhaps as an entire theory of processing. We leave this as an important but open problem.

Chapter 6. Conclusion

There is a natural tendency, now that we have succeeded in providing 3-LISP with reflective capabilities, for our analysis to shift from a study of what reflection is, to an investigation of how such facilities can best be used. Section 5.d identified a variety of open questions along these lines; further research is clearly mandated. In this last chapter, however, rather than pushing our inquiry forward, we will instead draw back from the details of 3-LISP and, by way of conclusion, will look briefly at the question of how 3-LISP — which is itself only an exemplar of a reflective formalism — fits into a larger conception of computation and semantics.

Several questions, in particular, arise in this regard. First, it is clear that rationalised semantics and procedural reflection are to some extent separable; this was manifested in our decision to present 2-LISP and 3-LISP as distinct dialects. We said at the outset that semantical *cleanliness* was a conceptual pre-requisite to reflection; we can now ask whether it would not be possible to retrofit a reflective capability into an evaluation-based dialect of LISP (we might imagine, for example, something called 2.7-LISP: a reflective version of 1.7-LISP, just as 3-LISP is a reflective version of 2-LISP). In other words, given that the semantical and reflective issues can be at least partially separated, is it our position that rationalised semantics is a *necessary* pre-requisite to reflection, or could reflective powers be developed in standard, non-reconstructed dialect?

For a variety of reasons this is not a question with a sharp yes/no answer, but we maintain our position that a *usable* reflective capability requires a semantically rationalised base, even if in some formal sense a procedurally reflective formalism could be built without such a base. There is no doubt that a "2.7-LISP style" dialect would be technically possible (especially now that 3-LISP can be used as a guide), but there are any number of reasons why it would be a bad idea, to the extent that the suggestion can even be made clear.

For one thing, in the rather philosophical analysis with which we started out, we defended our use of the term "reflection" because of the knowledge representation hypothesis, which made crucial reference to attributed declarative semantics. In fact, we

defined reflection in terms of what a process was reasoning *about*, and it is "aboutness" with which semantics is primarily concerned. Whether a procedural regimen counted as an instance of reflection would be judged, on our view, by looking at the semantical characterisation of it, including at the declarative fragment of that account. It would therefore be important to clarify, for any variant proposal, just what it claims regarding this pre-computational declarative attribution.

Specifically, there would be a question whether, in rejecting the semantical flatness of 2-LISP and retaining the notion of evaluation, one was in turn rejecting the entire account of attribution of declarative import, or whether one was *accepting* the account, but merely arguing for a procedural regimen (Ψ) that de-referenced sometimes (or even *always*[1]). One can clearly reject the *story* that we tell about declarative attribution (anything can be ignored), but one cannot, we maintain, reject the *phenomenon*; this is part of what chapter 3 is intended to argue. We have not *proposed* that people attribute declarative import to LISP structures, suggesting that it is wonderfully enlightening to take the numeral 3 as standing for the number three, and NIL (in traditional LISPs) as standing for Falsity. Rather, it is our claim — and it is a claim that it would surely be very hard to argue against — that we programmers *do* make just this kind of attribution. Therefore it is our view that the suggestion that one reject the declarative semantics amounts merely to a suggestion that our theoretical reconstruction of LISP pay no attention to how people understand LISP. Seen in this light, such a suggestion can be readily discounted.

Given then that one accepts the notion of an at least partly pre-computational semantics, what argument have we *against* an evaluation-based dialect? The substance of our argument was given in section 3.f.i; we need not repeat ourselves here. However there is another suggestion, not explored there, which is that we design our formalism so as *always* to de-reference (under such a proposal, in particular, it would *always* be the case that $\Psi(S) = \Phi(S)$). It is a consequence of such a view, of course, that one could never use the symbols T or NIL, or any numeral, in an evaluable context. Thus for example the expression

```
(+ 2 3)                                                    (S6-1)
```

would be semantically ill-formed. Rather, one would be required instead to use something of the following form:

```
(+ '2 '3)                                                    (S6-2)
```

Similarly with the boolean constants. Furthermore, it would seem that the symbol "+" would have to designate not the addition function, but the numeral-addition function (i.e. Φ("+") in this dialect would be what Ψ("+) is in 2-LISP and 3-LISP).

However there is a problem: a moment's thought leads one to realise that "+" could not designate *any function at all*, at least in a higher-order dialect (and we take it that the ability to "pass procedures as arguments" is a requirement; if we had to avoid that capability we would again simply dismiss the proposal as not serious). For suppose we defined a procedure that could meaningfully accept "+" as an argument:

```
(DEFINE A1                                  ; This is perfectly      (S6-3)
   (LAMBDA EXPR [FUN]                       ; acceptable in 2-LISP
      (LAMBDA EXPR [ARG] (FUN ARG 1))))     ; or 3-LISP.
```

Presumably the intent would be to support the following behaviour:

```
> ((A1 +) 7)                                                 (S6-4)
> 8
```

Or perhaps this (i.e., we are not currently concerned with the numerals):

```
> ((A1 +) '7)                                                (S6-5)
> 8
```

In the first line of S6-4 (and S6-5), the symbol "+" was evaluated (for want of another term, we continue to use this verb for the Ψ of this proposed dialect, although it is not clear that it signifies the evaluation we are used to), and by the current suggestion this means that it was de-referenced. Since functions *qua functions* are infinite, non-structural objects, it must follow that "+" does not designate a function, but rather some structural object (presumably something like a closure, of course, but the question is *what the notion of a closure comes to*, on this account).

This is all a little odd. LISP is ostensibly a functional language, but we have just been forced to admit that it cannot be used to deal with arithmetic (only with numerals), and we have now admitted that we cannot use terms that designate functions. The truth-values, sets, and all of mathematics would be dismissed for similar reasons. Nor do we even have the option of quoting the "functional terms", the way we quoted the numerals, as an escape; the following simply won't work (because of environment problems, as well as structural errors):

```
> ((A1 '+) '7)                                              (S6-6)
ERROR: Atoms cannot be applied.
```

In sum, if one accepts the suggestion that we abandon the notion of functions, as well as of numbers, then indeed it might be possible to construct a dialect that "always de-referenced". However it would appear that this move is merely a *reductio ad absurdum* of our own proposal. In particular, we can look at this proposal as if it embodied a decision merely to discard (or rather, to pay no attention to) what we have called Φ, and to name our Ψ relationship as one of "reference". Then *of course* Ψ "de-references" — but it does so *tautologically*, not for any interesting reason. In such a dialect one loses entirely the force of the question "*What is the semantical character of the function computed by the processor?*". One loses as well the subtlety of the relationships among Ψ, Φ, and Δ. Furthermore, the proposed semantics would not illuminate why the function (or procedure or closure or whatever) designated by the atom + happens to take the pair of numerals 3 and 4 onto the numeral 7 — a practice that can of course be defended only because those numerals designate numbers. All in all, it would appear that this proposal pays homage to a notion of reference that simply is not what we consider reference to be. Thus we will dismiss this suggestion as well.

There remains a third possibility: to accept our account of declarative semantics, and to adopt the semantically mixed notion of evaluation set out in the evaluation theorem. This is coherent — we have never denied that, nor do we have an argument that a procedurally reflective dialect of some sort could not be defined. All of our claims about how it would be difficult to understand, how it would fail to resonate with our tacit attribution, how it loses any claim to modelling true reflection, and so forth, would stand, but these are theoretical claims. A small piece of more practical evidence as to the difficulty of dealing with reflective procedures in such a scheme is provided by the MACLISP implementation of 3-LISP presented in the appendix. The task there is to encode, within MACLISP structures, an implementation of another dialect's structures. By and large 3-LISP structures are identified with MACLISP structures, as it happens (thus 3-LISP numerals are implemented as MACLISP numerals, 3-LISP atoms as MACLISP atoms, and so forth). Thus the MACLISP code is not dissimilar to the sort of reflective code one might imagine constructing in a reflective evaluation-based formalism. The code given in the appendix is replete with up-arrows and down-arrows, requiring considerable care to avoid making untenable

use/mention errors. Reflective code in 3-LISP, however, as the examples in the previous sections have shown, typically requires much less explicit level-crossing machinery. Thus, while we admit that a staunch advocate of evaluation could build an at least approximately reflective dialect, it is our contention that, although it might perhaps be initially easier to use (primarily because it would be more continuous with our LISP habits), as the complexity of programs increased, the dissonance between the procedural regimen and the naturally accorded semantics would defeat any serious attempts to use the reflective powers.

Another salient question has to do with the issue of adding reflective capabilities to programming languages other than LISP, and with the matter of rendering such languages semantically flat, in the spirit of 2-LISP. The first of these is straightforward, and was discussed very briefly in chapter 1; there would be no problem, providing a few requirements were met (providing an encoding of programs structures as valid data structures, formulating an explicit procedural theory of the language in the language, and so forth). The second is perhaps more interesting, especially as we move outside the realm of algebraic and functional paradigms, to more imperative and message-passing schemes, such as those manifested in FORTRAN or SMALLTALK. However even here it is clear that semantical flatness — and, even more simply, declarative import — would still make eminent sense. Consider for example the question of updating a display — a paradigmatically operational, rather than descriptive, type of behaviour. Even if the central *command* in such an interaction were defined primarily in terms of procedural import, however, the arguments with which it is phrased would presumably be cogent primarily declaratively. Suppose for example we wished to update the display corresponding to some particular editing buffer, and were led to write something of approximately the form DISPLAY(BUFFER(FILE_{16})). This "description" would most likely be simplified into a canonical (normal-form) name for the display in question — a process that would resemble our standard normalisation process, whether the procedural import was effected by command, or by procedure call, or by the passing of a message.

Furthermore, it is not our semantical line that *all* expressions be treated purely declaratively. This would be a fundamentalism of no particular merit (furthermore, it is a position that the λ-calculus and logic already explore); even natural languages like English are not *purely* declarative. The point, rather, is that those parts of the langauge that *are* declarative should be treated so; those that are procedural should be treated so; and the

interaction between them should be semantically sensible, from all points of view. The mere fact that we distinguish Ψ and Φ should not be taken as an argument that Φ is *better*, in any sense. Our claim is merely that *if* the declarative import (that Φ is an attempt to reconstruct) plays a role in how we observers understand the full significance of an computational expression, then it is best to design the language and its semantics so as to make sense of that attribution. No more; no less.

One final point deserves a moment's attention. Although our generic comments on reflection have not had this specific orientation, all of our technical work has focused on the provision of reflection in *programming languages*, not in computationally based processes defined or constructed in such languages. The concept of reflection is in no way restricted to language design; it is easy to imagine, for example, a natural-language query system for a data base, or any other process, designed with reflective capabilities, so that it could stop at any point and deal reasonably with its interaction with the world, and with its own internal state. We have taken the approach we have for two reasons. First, the *details* of any such use have more to do with the question of how to *use* reflection, rather than how to provide it, and as such they stand as open questions for further research. Second, we hold that a coherent treatment of reflection of the sort presented here is a *pre-requisite* to any such exploration; without it, any attempt to construct specific reflective systems would founder for lack of a theoretical base. The basic distinctions and results we have explored, furthermore, and the underlying architecture of 3-LISP, should carry over intact into a particular situation. In other words, although 3-LISP is indeed a programming language, the understanding of reflection that it is intended to embody is not specific to programming languages *per se*; it should serve in more particular situations equally well.

There is of course a reason for this last point, arising from our constant methodological stance. Our theories of reflection and of semantics have been derived almost entirely from intuitions and insights into our natural human use of thought and language, not from programming languages viewed as an isolated phenomenon. In fact it is one of our foundational assumptions that computational concepts in general — certainly including programming languages — are, in the deepest sense, derivative on our understanding of mind. It has therefore been our intent to apply an understanding of language to an ostensibly computational matter, with the hope, in part, of integrating the theoretical frameworks of the two disciplines. There seems no doubt that theories of

language can help in understanding computational issues; whether the contribution in the other direction will prove as useful only time will tell. On the surface it seems at least plausible that the theories we have articulated here might play a role in our understanding of human language, but these are speculations for a different time and place. For now our investigation remains computational; authentic human reflection is a much larger question.

Appendix. A MACLISP Implementation of 3-LISP

The code listed in the following pages was printed out from a version of the 3-LISP implementation running on the CADR processor (a version of the M.I.T. LISP machine) at the Xerox Palo Alto Research Center, on January 23, 1982. This implementation was originally constructed in MACLISP, and was modified minimally to run on the LISP machines in the fall of 1982; the only changes were made to the input/output and interrupt routines, which could be readily changed back to a MACLISP format if necessary. The code is by and large documented; the intent, as mentioned at the outset, was merely to demonstrate as transparently as possible the functionality of the 3-LISP abstract virtual machine; as a consequence the performance of this implementation is unacceptably slow for anything except small examples. The construction of a fast but full implementation of 3-LISP is a project that should be undertaken in the near future.

```
;;; -*- Mode:LISP; Package:User; Base: 10. -*-                    Page 1
;;;
;;;                              3-LISP
;;;                              ======
;;;
;;; A statically scoped, higher order, semantically rationalised, procedurally
;;; reflective dialect of LISP, supporting SIMPLE and REFLECTIVE procedures.
;;;
;;; This is a straightforward and EXTREMELY INEFFICIENT implementation; the
;;; intent is merely to manifest the basic 3-LISP functionality.  A variety
;;; of techniques could increase the efficiency by several orders of magnitude
;;; (most obvious would be to avoid consing explicit continuation structures at
;;; each step of NORMALISE).  With some ingenuity 3-LISP could be implemented
;;; as efficiently as any other dialect.
;;;
;;; 1. Structural Field:
;;; --------------------
;;;
;;;     Structure Type     Designation             Notation
;;;
;;;     1. Numerals     -- Numbers              -- sequence of digits
;;;     2. Booleans     -- Truth values         -- $T or $F
;;;     3. Pairs        -- Functions (& appns)  -- (<exp> . <exp>)
;;;     4. Rails        -- Sequences            -- [<exp> <exp> ... <exp>]
;;;     5. Handles      -- S-expressions        -- '<exp>
;;;     6. Atoms        -- (whatever bound to)  -- sequence of alphanumerics
;;;
;;; a. There is no derived notion of a LIST, and no atom NIL.
;;; b. Pairs and rails are pseudo-composite; the rest are atomic.
;;; c. Numerals, booleans, and handles are all normal-form and canonical.
;;;    Some rails (those whose elements are normal form) and some pairs
;;;    (the closures) are normal form, but neither type is canonical.
;;;    No atoms are normal-form.
;;;
;;; 2. Semantics:  The semantical domain is typed as follows:
;;; ------------
;;;                                          ___ numeral
;;;                                         |___ boolean
;;;                    ____ s-expression ___|___ pair
;;;                   |                     |___ rail
;;;                   |                     |___ handle
;;;                   |                     |___ atom
;;;        Object ____|
;;;                   |                     ___ number
;;;                   |____ abstraction ___|___ truth-value
;;;                   |                     |___ sequence
;;;                   |
;;;                   |_____________________ function
;;;
;;; 3. Notation:
;;; ------------
;;;
;;; Each structural field category is notated with a distinguishable notational
;;; category, recognisable in the first character, as follows (thus 3-LISP
;;; could be parsed by a grammar with a single-character look-ahead):
;;;
;;;     1. Digit        -->  Numeral      4. Left bracket    -->  Rail
;;;     2. Dollar Sign  -->  Boolean      5. Singe quote     -->  Handle
;;;     3. Left paren   -->  Pair         6. Non-digit       -->  Atom
;;;
;;; The only exceptions are that numerals can have a leading "+" or "-", and in
;;; this implementation an atom may begin with a numeral providing it contains
;;; at least one non-digit (since MACLISP supports that).
```

```
064   ;;;
065   ;;; BNF Grammar  Double quotes surround object level constants, "←" indicates
066   ;;; -----------  concatenation, brackets delineate groupings, "*" means
067   ;;;              zero-or-more repetition, and "|" separates alternatives:
068   ;;;
069   ;;; formula     ::= [break←]* form [←break]*
070   ;;; form        ::= L-numeral | L-boolean | L-pair | L-rail | L-handle | L-atom
071   ;;;
072   ;;; L-numeral   ::= ["+"← | "-"←]* digit [←digit]*
073   ;;; L-boolean   ::= "$T" | "$F"
074   ;;; L-pair      ::= "("← formula ←"."← formula ←")"
075   ;;; L-rail      ::= "["← [formula←]* "]"
076   ;;; L-handle    ::= "'"← formula
077   ;;; L-atom      ::= [character←]* non-digit [←character]*
078   ;;;
079   ;;; character   ::= digit | non-digit
080   ;;; non-digit   ::= alphabetic | special
081   ;;;
082   ;;; digit       ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "0"
083   ;;; alphabetic  ::= "a" | "b" | "c" | ... | "A" | "B" | "C" | ... etc.
084   ;;; special     ::= "*" | "-" | "+" | "/" | "@" | "#" | "%" | "&" | "<" | ">" |
085   ;;;                 "←" | "=" | "\" | "?" | ":" | "~" | "!"
086   ;;; reserved    ::= "'" | ";" | "(" | ")" | "[" | "]" | "{" | "}" | "|" | """ |
087   ;;;                 "," | "." | "↑" | "`" | "$" | <space> | <end-of-line>
088   ;;;
089   ;;; break       ::= <space> | <end-of-line> | comment
090   ;;; comment     ::= ";" [←character | ←reserved | ←<space> ]* <end-of-line>
091   ;;;
092   ;;; The Lexical Notation Interpretation Function THETA (by category):
093   ;;; ---------------------------------------------------------------
094   ;;;
095   ;;; L-numeral   -- Numerals in the standard fashion;
096   ;;; L-boolean   -- $T and $F to each of the two booleans;
097   ;;; L-pair      -- A new (otherwise inaccessible) pair whose CAR is THETA of
098   ;;;                the first formula and whose CDR is THETA of the second;
099   ;;; L-rail      -- A new (otherwise inaccessible) rail whose elements are THETA
100   ;;;                of each of the constituent formulae;
101   ;;; L-handle    -- The handle of THETA of the constituent formula.
102   ;;; L-atom      -- The corresponding atom.
103   ;;;
104   ;;; NOTES:
105   ;;;
106   ;;; 1. Case is ignored (converted to upper case on input)
107   ;;; 2. Notational Sugar:
108   ;;;
109   ;;;     "(<e1> <e2> ... <en>)" abbreviates "(<e1> . [<e2> ... <en>])"
110   ;;;
111   ;;; 3. We use exclamation point in place of down-arrow, since MACLISP does
112   ;;;    not support the latter character (it is not in ASCII, sadly).
113   ;;; 4. A Summary of the use of reserved characters:
114   ;;;
116   ;;;    a:  (  -- starts pairs                h:  .  -- in "[ ... ]" for JOIN
116   ;;;    b:  )  -- ends pairs                  i:  ↑  -- NAME
117   ;;;    c:  .  -- in "( ... )" for CDR        j:  !  -- REFERENT
118   ;;;    d:  [  -- starts rails               (k:  :  -- DYNAMIC)
119   ;;;    e:  ]  -- ends rails                  l:  `  -- Backquote a la MACLISP
120   ;;;    f:  '  -- starts handles             m:  ,  --    "      "  "    "
121   ;;;    g:  ;  -- starts comments (to CRLF)   n:  ~  -- Switch to MACLISP
122   ;;;
123   ;;;    A-g are primitive, h-m are sugar, and n is implementation-specific.  In
124   ;;;    this implementation, since "!" is used for REFERENT (it should be
125   ;;;    down-arrow), it is reserved rather than special.  Similarly, "~" is
126   ;;;    reserved in this implementation for the MACLISP escape.  Finally, the
127   ;;;    characters "{", "}", "|", and """ are reserved but not currently used
128   ;;;    (intended for sacks, arbitrary atom names (a la MACLISP) and strings).
```

```
129                                                                    Page 1:2
130   ;;; 4. Processor:
131   ;;; -------------
132   ;;;
133   ;;; The main driving loop of the processor is a READ-NORMALISE-PRINT loop
134   ;;; (see item 6, below), taking expressions into normal-form co-designators.
135   ;;; The normal form designators for each of the semantic types are:
136   ;;;
137   ;;;        Semantic type                  Normal form designator (NFD)
138   ;;;
139   ;;;     1. Numbers                        Numerals
140   ;;;     2. Truth-values                   Boolean constants
141   ;;;     3. S-expressions                  Handles
142   ;;;     4. Sequences                      Rails of NFD's of the elements
143   ;;;     5. Functions                      Pairs: (<type> <env> <pattern> <body>)
144   ;;;     6. Environments                   Rails: [['<a1> '<b1>] ['<a2> '<b2>] ... ]
145   ;;;
146   ;;; 1-3 are CANONICAL, 4-6 are not.  Thus, A = B implies ↑A = ↑B only if A and
147   ;;; B designate numbers, truth-values, or s-expressions.
148
149   ;;; 5. Primitive procedures:
150   ;;; ------------------------
151   ;;;
152   ;;;     Summary (fuller definitions are given below):
153   ;;;
154   ;;; Typing:       TYPE                    -- defined over 10 types (4 syntactic)
155   ;;; Identity:     =                       -- defined over s-expressions, truth-
156   ;;;                                          values, sequences, and numbers
157   ;;; Structural:   PCONS, CAR, CDR         -- to construct and examine pairs
158   ;;;               LENGTH, NTH, TAIL       -- to examine rails and sequences
159   ;;;               RCONS, SCONS, PREP      -- to construct  "   "      "
160   ;;; Modifiers:    RPLACA, RPLACD          -- to modify pairs
161   ;;;               RPLACN, RPLACT          -- "    "    rails
162   ;;; Functions:    SIMPLE, REFLECT         -- make procedures from expressions
163   ;;; Control:      EF                      -- an extensional if-then-else conditional
164   ;;; Semantics:    NAME, REFERENT          -- to mediate between sign & signified
165   ;;; Arithmetic:   +, -, *, /              -- as usual
166   ;;; I/O:          READ, PRINT, TERPRI     -- as usual
167   ;;; Reflection:   LEVEL                   -- the current reflective level
168   ;;;
169   ;;; The following kernel functions need NOT be primitive; they are defined in
170   ;;; the reflective model in terms of the above:
171   ;;;
172   ;;;     DEFINE, LAMBDA, NORMALISE, REDUCE, SET, BINDING, MACRO
173   ;;;
174   ;;; Syntax and definitions:
175   ;;;
176   ;;;     Form of use              Designation (environment relative):
177   ;;;
178   ;;;  (TYPE <exp>)             -- The atom indicating the type of <exp> (one of
179   ;;;                              the 10 on the fringe of the tree in #2, above)
180   ;;;
181   ;;;  (= <a> <b>)              -- Truth if <a> and <b> are the same, falsity
182   ;;;                              otherwise, providing <a> and <b> are of the
183   ;;;                              same type, and are s-expressions, truth-values,
184   ;;;                              sequences, or numbers
185   ;;;
186   ;;;  (PCONS <a> <b>)          -- A (new) pair whose CAR is <a> and CDR is <b>
187   ;;;  (CAR <a>)                -- The CAR of pair <a>
188   ;;;  (CDR <a>)                -- The CDR of pair <a>
189   ;;;  (RPLACA <a> <b>)         -- The new CAR <b> of modified pair <a>
190   ;;;  (RPLACD <a> <b>)         -- The new CDR <b> of modified pair <a>
191   ;;;
192   ;;;  (LENGTH <a>)             -- The length of rail or sequence <a>
193   ;;;  (NTH <n> <a>)            -- The <n>th element of rail or sequence <a>
194   ;;;  (TAIL <n> <a>)           -- Tail of rail/seq <a> starting after <n>th elemnt
195   ;;;  (RCONS <a1> ... <ak>)    -- A new rail whose elements are <a1>, ... , <ak>
196   ;;;  (SCONS <a1> ... <ak>)    -- The sequence whose elements are <a1>, ..., <ak>
197   ;;;  (PREP <a> <rs>)          -- A new rail/seq whose 1st is <a>, 1st tail is <b>
198   ;;;  (RPLACN <n> <a> <b>)     -- The new <n>th element <b> of modified rail <a>
199   ;;;  (RPLACT <n> <a> <b>)     -- The new <n>th tail <b> of modified rail <a>
```

```
200   ;;;
201   ;;;  (SIMPLE  <e> <p> <b>)  -- NOT FOR CASUAL USE! (The function of given type
202   ;;;  (REFLECT <e> <p> <b>)     designated by the lambda abstraction of pattern
203   ;;;                            <p> over expression <b> in environment <e>)
204   ;;;
205   ;;;  (EF <p> <a> <b>)        -- <a>, if <p> designates truth; <b> if falsity.
206   ;;;
207   ;;;  (NAME <a>)              -- The (or a) normal-form designator of <a>
208   ;;;  (REFERENT <a> <env>)    -- The object designated by <a> in environment <env>
209   ;;;
210   ;;;  (+ <a> <b>)             -- The sum, difference, produce, and quotient of
211   ;;;  (- <a> <b>)                <a> and <b>, respectively
212   ;;;  (* <a> <b>)
213   ;;;  (/ <a> <b>)
214   ;;;
215   ;;;  (READ)                  -- The s-expression notated by the next formula in
216   ;;;                             the input stream.
217   ;;;  (PRINT <a>)             -- <a>, which has just been printed.
218   ;;;
219   ;;;  (LEVEL)                 -- The number of the current reflective level.
220
221   ;;; 6. Processor Top Level:
222   ;;; ------------------------
223   ;;;
224   ;;; Each reflective level of the processor is assumed to start off
225   ;;; running the following function:
226   ;;;
227   ;;;   (define READ-NORMALISE-PRINT
228   ;;;      (lambda simple [env]
229   ;;;         (block (prompt (level))
230   ;;;                (let [[normal-form (normalise (read) env id)]]
231   ;;;                   (prompt (level))
232   ;;;                   (print normal-form)
233   ;;;                   (read-normalise-print env)))))
234   ;;;
235   ;;; The way this is imagined to work is as follows: the very top processor
236   ;;; level (infinitely high up) is invoked by someone (say, God, or some
237   ;;; functional equivalent) normalising the expression (READ-NORMALISE-PRINT
238   ;;; GLOBAL).  When it reads an expression, it is given the input string
239   ;;; "(READ-NORMALISE-PRINT GLOBAL)", which causes the level below it to read
240   ;;; an expression, which is in turn given "(READ-NORMALISE-PRINT GLOBAL)",
241   ;;; and so forth, until finally the second reflective level is given
242   ;;; "(READ-NORMALISE-?RINT GLOBAL)".  This types out "1>" on the console,
243   ;;; and awaits YOUR input.
244   ;;;
245   ;;; 7. Environments:
246   ;;; ----------------
247   ;;;
248   ;;; Environments are sequences of two-element sequences, with each sub-sequence
249   ;;; consisting of a variable and a binding (both of which are of course
250   ;;; expressions).  A normal-form environment designator, therefore, is a rail of
251   ;;; rails, with each rail consisting of two handles.  Variables are looked up
252   ;;; starting at the front (i.e. the second element of the first subrail whose
253   ;;; first element is the variable is the binding of that variable in that
254   ;;; environment).  Environments can also share tails: this is implemented by
255   ;;; normal-form environment designators sharing tails (this is used heavily in
256   ;;; the GLOBAL/ROOT/LOCAL protocols, and so forth).  Effecting a side-effect on
257   ;;; the standard normal-form environment designator CHANGES what the environment
258   ;;; is, which is as it should be.  Each level is initialised with the same global
259   ;;; environment (the implementation does not support root environments -- see
260   ;;; note 11).
```

```
261
262   ;;; 8. Implementation:
263   ;;; ------------------
264   ;;;
265   ;;;     3-LISP Structural Type:    MACLISP implementation:
266   ;;;
267   ;;;     1. Numerals              -- Numerals
268   ;;;     2. Booleans              -- The atoms $T and $F
269   ;;;     3. Pairs                 -- Pairs
270   ;;;     4. Rails                 -- (~RAIL~ <e1> ... <en>)  (but see note 9)
271   ;;;     5. Handles               -- (~QUOTE~ . <exp>)
272   ;;;     6. Atoms                 -- atoms (except for $T, $F, ~RAIL~, ~QUOTE~,
273   ;;;                                 ~C0~, ~C1~, ~C2~, ~C3~, ~C4~, ~C5~, ~PRIM~,
274   ;;;                                 and NIL)
275   ;;;
276   ;;; The main processor functions constantly construct MACLISP representations
277   ;;; of the 3-LISP normal-form designators of the continuations and environments
278   ;;; that WOULD be being used if the processor were running reflectively.  In
279   ;;; this way functions that reflect can be given the right arguments without
280   ;;; further ado.  In assembling these continuations and environments (see
281   ;;; 3-NORMALISE etc.), the code assumes that the incoming values are already in
282   ;;; normal form.  A more efficient but trickier strategy would be to put these
283   ;;; objects together only if and when they were called for; I haven't attempted
284   ;;; that here.  This would all be made simpler if both environments and
285   ;;; continuations were functions abstractly defined: no copying of structure
286   ;;; would ever be needed, since the appropriate behaviour could be wrapped
287   ;;; around the information in whatever form it was encoded in the primitive
288   ;;; implementation.
289   ;;;
290   ;;; Two major recognition strategies are used for efficiency.  Those instances
291   ;;; of the four STANDARD continuation types that were generated by the MACLISP
292   ;;; version of the processor are trapped and decoded primitively: if this were
293   ;;; not done the processor would reflect at each step.  Also, explicit calls to
294   ;;; REDUCE and NORMALISE are trapped and run directly by the implementing
295   ;;; processor: this is not strictly necessary, but unless it were done the
296   ;;; processor might never come down again after reflecting up.
297   ;;;
298   ;;; The standard continuation types, called C0 - C3, are identified in the
299   ;;; comments and in the definitions of NORMALISE and REDUCE (q.v.), and listed
300   ;;; below.  These types must be recognized by 3-APPLY and 3-REDUCE, so that the
301   ;;; implementing processor can drop down whenever possible, whether or not the
302   ;;; explicit interpretation of a (non-primitive) reflective function has
303   ;;; intervened.  The atoms ~C0~, ~C1~, ~C2~, and ~C3~ -- called the SIMPLE
304   ;;; ALIASES -- are used instead of the primitive SIMPLE closure as the function
305   ;;; type (i.e. as the CAR of the continuation closures).  These atoms are also
306   ;;; MACLISP function names to effect the continuation).  The implementation
307   ;;; makes these atoms look = to the SIMPLE closure, so that the user cannot
308   ;;; tell different atoms are being used, but so that the continuations can be
309   ;;; trapped.
310   ;;;
311   ;;; Three other simple aliases are used (~C4~, ~C5~, and ~PRIM~).  ~C4~ is used
312   ;;; to identify the continuation used by READ-NORMALISE-PRINT, since the higher
313   ;;; level READ-NORMALISE-PRINT continuation may not explicitly exist.  ~C5~ is
314   ;;; used by the IN-3-LISP macro to read in 3-LISP code embedded within MACLISP
315   ;;; (it can therefore be used to read in 3-LISP code in files and so forth).
316   ;;; ~PRIM~ is used in normal-form designators of primitive procedures.  Thus,
317   ;;; while PCONS in the initial global environment looks to a 3-LISP program to
318   ;;; normalise to (<SIMPLE> '[ ... <global>] '[A B] '(PCONS A B)), in fact the
319   ;;; CAR of that form is ~PRIM~, not <SIMPLE>.
320   ;;;
321   ;;; The four standard continuations:
322   ;;;
323   ;;;    C0:  Accept the normalised function designator in an application.
324   ;;;    C1:  Accept the normalised arguments for a SIMPLE application.
325   ;;;    C2:  Accept the normalised first element in a rail fragment.
326   ;;;    C3:  Accept the normalised tail of a rail fragment.
327   ;;;
328   ;;;   (C4:  Identifies top level call of READ-NORMALISE-PRINT.)
329   ;;;   (C5:  Used in order to read in 3-LISP structures by IN-3-LISP.)
```

```
331   ;;; Programming conventions:
332   ;;;
333   ;;; Special variables are prefixed with "3-".  Procedures are prefixed with "3-".
334   ;;; If they operate on MACLISP structures implementing 3-LISP structures, the
335   ;;; procedure name is defined with respect to the operation viewed with respect
336   ;;; to the 3-LISP structure.  For example, 3-EQUAL returns T if the two arguments
337   ;;; encode the same 3-LISP structure.
338   ;;;
339   ;;; NOTE: In fall 1981, the implementation was minimally changed to run on an MIT
340   ;;; CADR machine, not in MACLISP.  The only concessions to the new base were in
341   ;;; the treatment of I/O and interrupts; no particular features of the CADR have
342   ;;; been used.  It should therefore require minimal work to retrofit it to a
343   ;;; MACLISP base.
344
345   ;;; 9. Rails: Implementation and Management:
346   ;;; ----------------------------------------
347   ;;;
348   ;;; The implementation of rails is tricky, because RPLACT modifications must be
349   ;;; able to take effect on the 0'th tail, as well as subsequent ones, requiring
350   ;;; either the use of full bi-directional linkages, or "invisible pointers" (a
351   ;;; true LISP-machine implementation could perhaps use the underlying invisible
352   ;;; pointer facility) and special circularity checking.  We choose the latter
353   ;;; option.  The implementation (where "+" means one or more, "*" means zero or
354   ;;; more) of a rail is:
355   ;;;
356   ;;;     [a b ... z]   ==>   (<~RAIL~>+ a <~RAIL~>* b ... <~RAIL~>* z <~RAIL~>*)
357   ;;;
358   ;;; where the ~RAIL~ atoms are effectively invisible, but begin every rail that
359   ;;; is given out to the outside world (and can thus be used to distinguish
360   ;;; rails from 3-LISP cons pairs).  Just reading in [A B ... Z] generates
361   ;;; (~RAIL~ A B ... Z).
362   ;;;
363   ;;; Unless RPLACT's are done, the number of ~RAIL~ atoms cannot exceed the number
364   ;;; of elements.  With arbitrary RPLACT'ing, the efficiency can get arbitrarily
365   ;;; bad (although it could be corrected back to a linear constant of 2 by a
366   ;;; compacting garbage collector.)
367   ;;;
368   ;;; 10. User Interface:
369   ;;; -------------------
370   ;;;
371   ;;; To run 3-LISP, load the appropriate one of the following FASL files:
372   ;;;
373   ;;;     ML:     ML:BRIAN;3-LISP FASL
374   ;;;     PARC:   [Phylum]<BrianSmith>3-lisp>3-lisp.qfasl
375   ;;;
376   ;;; The processor can be started up by executing (3-LISP), and re-initialised
377   ;;; completely at any point by executing (3-INIT) (both in MACLISP).  The
378   ;;; READ-NORMALISE-PRINT loop prints the current reflective level to the left
379   ;;; of the prompt character.  The following interrupt characters are defined:
380   ;;;
381   ;;;     a. Control-E    -- Toggles between MACLISP and 3-LISP.
382   ;;;
383   ;;;     b. Control-G    -- Quit to level 1       (regular quit in MACLISP)
384   ;;;     c. Control-F    -- Quit to current level (regular quit in MACLISP)
385   ;;;
386   ;;; To read in and manipulate files, surround an arbitrary number of
387   ;;; expressions with the MACLISP wrapping macro IN-3-LISP, and precede each
388   ;;; 3-LISP expression with a backslash, so that it will be read in by the
389   ;;; 3-LISP reader.  Then load the file as if it were a regular MACLISP file.
390   ;;; For example:
391   ;;;
392   ;;;     (in-3-lisp
393   ;;;       \(define increment (lambda simple [x] (+ x 1)))
394   ;;;       \(define quit      (lambda reflect [] 'QUIT)))
395   ;;;
396   ;;; Equivalent, and with the advantage that TAGS and @ see the definitions, is:
397   ;;;
398   ;;;     (in-3-lisp \[
399   ;;;
400   ;;;     (define increment (lambda simple [x] (+ x 1)))
401   ;;;     (define quit      (lambda reflect ? 'QUIT))                ])
```

```
404   ;;;
405   ;;; 11. Limitations of the Implementation:
406   ;;; ---------------------------------------
407   ;;;
408   ;;; There are a variety of respects in which this implementation is incomplete
409   ;;; or flawed:
410   ;;;
411   ;;; 1. Side effects to the reflective procedures will not be noticed -- in a
412   ;;;    serious implementation these procedures would want to be kept in a pure
413   ;;;    page so that side effects to them could be trapped, causing one level
414   ;;;    of reflective deferral.
415   ;;;
416   ;;; 2. Reflective deferral is not yet support at all.  No problems are
417   ;;;    expected; it merely needs attention.
418   ;;;
419   ;;; 3. In part because I think it may be a bad idea, this implementation does
420   ;;;    not support a root environment protocol.
421   ;;;
422   ;;; 12. Obvious Extensions:
423   ;;; -----------------------
424   ;;;
425   ;;; Obvious extensions to the implementation fall into two groups: those that
426   ;;; would increase the efficiency of the implementation, but not change its
427   ;;; basic functionality, and those that would extend that functionality.
428   ;;; Regarding the first, the following are obvious candidates:
429   ;;;
430   ;;; 1. Get rid of the automatic consing of continuation and environment
431   ;;;    structures, as mentioned earlier.
432   ;;;
433   ;;; 2. Support various intensional procedures (LAMBDA, IF, COND, MACRO, SELECT,
434   ;;;    and so forth) as primitives.  This would require the virtual provision
435   ;;;    of all of the continuation structure at the reflective level that would
436   ;;;    have been generated had the definitions used here been used explicitly:
437   ;;;    it wouldn't be trivial.  Unless, of course, the language was redefined
438   ;;;    to include these as primitives (but the current proof of its finiteness
439   ;;;    depends on no reflective primitives, so this too would take some work).
440   ;;;
441   ;;; Functional extensions include:
442   ;;;
443   ;;; 1. Make the bodies of LAMDBA, LET, COND, etc. take multiple expressions
444   ;;;    (i.e. be virtual BLOCK bodies).
445   ;;;
446   ;;; 2. Strings (and normal-form string designators, perhaps called "STRINGERS")
447   ;;;    could be added.
448
```

```
001
002   ;;; Declarations and Macros:
003   ;;; ------------------------
004
005   (declare
006     (special
007       3=simple-aliases 3=global-environment 3=states 3=level 3=break-flag
008       3=in-use 3=readtable L=readtable S=readtable 3=a1 3=a2 3=a3 3=a4
009       3=normalise-closure 3=reduce-closure 3=simple-closure 3=reflect-closure
010       3=id-closure 3=backquote-depth ignore 3=process)
011     (*lexpr 3-read 3-read* 3-error))
012
013   ;;; (herald 3-LISP)
014
015   (eval-when (load eval compile)
016
017   (defmacro list? (x) `(eq (typep ,x) 'list))
018   (defmacro 1st (l) `(car ,l))
019   (defmacro 2nd (l) `(cadr ,l))
020   (defmacro 3rd (l) `(caddr ,l))
021
022   )
023
024   (defmacro 3-primitive-simple-id  (proc) `(cadr (3r-3rd (cdr ,proc))))
025
026   (defmacro 3-numeral (e) `(fixp ,e))
027   (defmacro 3-boolean (e) `(memq ,e '($T $F)))
028
029   (defmacro 3-bind (vars vals env)
030      `(cons '~RAIL~ (nconc (3-bind* ,vars ,vals) ,env)))
031
032   ;;; Two macros having to do with input:
033
034   (defmacro in-3-lisp (&rest body)
035      `(progn (or (boundp '3=global-environment) (3-init))
036              ,@(do ((exprs body (cdr exprs))
037                     (forms nil (cons `(3-lispify ',(car exprs)) forms)))
038                    ((null exprs) (nreverse forms)))))
039
040   (defmacro ~3-BACKQUOTE (expr) (3-expand expr nil))
041
042   ;;; 3-NORMALISE*   If MACLISP were tail-recursive, calls to this would
043   ;;; ------------   simply call 3-NORMALISE.  Sets up the loop variables
044   ;;;                and jumps to the top of the driving loop.
045
046   (defmacro 3-normalise* (exp env cont)
047      `(progn (setq 3=a1 ,exp 3=a2 ,env 3=a3 ,cont)
048              (*throw '3-main-loop 'nil)))
049
050   ;;; The rest of the macro definitions are RAIL specific:
051
052   (defmacro 3r-1st (exp) `(car (3-strip ,exp)))
053   (defmacro 3r-2nd (exp) `(car (3-strip (3-strip ,exp))))
054   (defmacro 3r-3rd (exp) `(car (3-strip (3-strip (3-strip ,exp)))))
055   (defmacro 3r-4th (exp) `(car (3-strip (3-strip (3-strip (3-strip ,exp))))))
056
057   ;;; Macros for RAIL management:
058
059   ;;; 3-STRIP           -- Returns a rail with all ~RAIL~ headers removed.  Have
060   ;;; -------              have to step through as many headers as have built up.
061   ;;;
062   ;;; 3-STRIP*          -- Returns the last header of arg -- used for RPLACD, and
063   ;;; --------             to establish rail identity.  Steps down through headers.
064
065   (eval-when (load eval compile)
066
067   (defmacro 3-strip (rail)
068      `(do ((rest (cdr ,rail) (cdr rest)))
069           ((not (eq (car rest) '~RAIL~)) rest)))
```

```
070
071   (defmacro 3-strip* (rail)
072     `(do ((rest ,rail (cdr rest)))
073          ((not (eq (cadr rest) '~RAIL~)) rest)))
074
075   )
076
077   ;;; 3-LENGTH*        -- Return the length of a 3-LISP rail.
078
079   (defmacro 3-length* (rail)
080     `(do ((n 0 (1+ n))
081           (rail (3-strip ,rail) (3-strip rail)))
082          ((null rail) n)))
083
```

```
001
002   ;;; Input/Output:
003   ;;; -------------
004   ;;;
005
006   ;;; A special readtable (3-READTABLE) is used to read in 3-LISP notation, since
007   ;;; it must be parsed differently from MACLISP notation.  The 3-LISP READ-
008   ;;; NORMALISE-PRINT loop uses this; in addition, a single expression will be
009   ;;; read in under the 3-LISP reader if preceded by backslash ("\") in the
010   ;;; MACLISP reader.  Similarly, a single expression will be read in by the
011   ;;; MACLISP reader if preceded with a tilde ("~") in the 3-LISP reader.
012   ;;;
013   ;;; MACLISP and 3-LISP both support backquote.  The readers and the backquotes
014   ;;; can be mixed, but be cautious: the evaluated or normalised expression must
015   ;;; be read in with the right reader.  For example, a MACLISP backquoted
016   ;;; expression can contain a 3-LISP fragment with a to-be-evaluated-by-MACLISP
017   ;;; constituent, but a tilde is required before it, so that the MACLISP reader
018   ;;; will see it.  Example: "`\[value ~,(plus x y)]".  ",@" and ",." are not
019   ;;; supported by the 3-LISP backquote.
020   ;;;
021   ;;; Any 3-LISP backquoted expression will expand to a new-structure-creating
022   ;;; expression at the level of the back-quote, down to and including any level
023   ;;; including a comma'ed expression.  Thus `[] expands to (rcons), `[[a b c] [d
024   ;;; ,e f]] expands to (rcons '[a b c] (rcons 'd e 'f)), and so forth.  This is
025   ;;; done so as to minimise the chance of unwanted shared tails, but to avoid
026   ;;; unnecessary structure consing.  We use `[] in place of (rcons) many times in
027   ;;; the code.
028   ;;;
029   ;;; Expressions like "~~CO~" are necessary in order to get the aliases into
030   ;;; 3-LISP, since the first tilde flips readers.  Once 3-LISP has been
031   ;;; initialised the aliases will be rejected: to reload a function containing an
032   ;;; alias, temporarily bind 3-simple-aliases to NIL.
033   ;;;
034   ;;; There are two special read macro characters, for name and referent (MACLISP
035   ;;; and 3-LISP versions).  (Ideally these would be uparrow and downarrow, but
036   ;;; down-arrow is unfortunately not an ASCII character):
037   ;;;
038   ;;;       Form            MACLISP expansion       3-LISP expansion
039   ;;;
040   ;;;      1. ↑<exp>        (3-NAME <exp>)          (NAME <exp>)
041   ;;;      2. !<exp>        (3-REF <exp>)           (REFERENT <exp> (current-env))
042
043   (eval-when (load eval compile)
044
045   ;;; Five constants need to be defined for 3-LISP structures to be read in:
046
047   (setq S-readtable readtable                       ; Save the system readtable
048         L-readtable (copy-readtable)                ; and name two special ones:
049         3-readtable (copy-readtable)                ; one for LISP, one for 3-LISP.
050         3-simple-aliases nil                        ; Make these NIL so we can read
051         3-backquote-depth 0)                        ; in the aliases in this file!
052
053   ;;; The following has been modified from the original MACLISP to enable it to
054   ;;; operate under the I/O protocols of the MIT LISP machine:
055
056   (login-setq readtable L-readtable)                ; Needed in order to read this file.
057
058   (let ((readtable L-readtable))
059     (set-syntax-macro-char #/\ #'(lambda (1 s) (3-read s)))
060     (set-syntax-macro-char #/↑ #'(lambda (1 s) `(cons '~QUOTE~ ,(read s))))
061     (set-syntax-macro-char #/! #'(lambda (1 s) `(3-ref ,(read s))))
062     (set-syntax-from-description #/] 'si:single))       ; So "~FOO]" will work.
```

```
063
064   (let ((readtable 3-readtable))
065      (set-syntax-macro-char #/~ #'(lambda (l s) (let ((readtable L-readtable)) (read s))))
066      (set-syntax-macro-char #/| #'(lambda (l s) `(referent ~RAIL~ ,(3-read* s)
067                                                            (current-env ~RAIL~))))
068      (set-syntax-macro-char #/↑ #'(lambda (l s) `(name ~RAIL~ ,(3-read* s))))
069      (set-syntax-macro-char #/' #'(lambda (l s) `(~QUOTE~ . ,(3-read* s))))
070      (set-syntax-macro-char #/( #'(lambda (l s) (3-read-pair s)))
071      (set-syntax-macro-char #/[ #'(lambda (l s) (3-read-rail s)))
072      (set-syntax-macro-char #/` #'(lambda (l s) (3-backq-macro s)))
073      (set-syntax-macro-char #/, #'(lambda (l s) (3-comma-macro s)))
074      (set-syntax-from-description #/) 'si:single)
075      (set-syntax-from-description #// 'si:single)
076      (set-syntax-from-description #/$ 'si:single)
077      (set-syntax-from-description #/] 'si:single)
078      (set-syntax-from-description #/. 'si:single))
079
080   ;;; 3-READ(*)  Read in one 3-LISP s-expression (*-version assumes the
081   ;;; ---------  3-LISP readtable is already in force, and accepts an
082   ;;;            optional list of otherwise illegal atoms to let through).
083
084   (defun 3-read (&optional stream)
085      (let ((readtable 3-readtable)) (3-read* stream)))
086
087   (defun 3-read* (stream &optional OK)
088      (let ((token (read stream)))
089         (cond ((memq token OK) token)
090               ((memq token '(|)| |.| |]|)) (3-illegal-char token))
091               ((or (memq token '(~RAIL~ ~QUOTE~ NIL))
092                    (memq token 3-simple-aliases)) (3-illegal-atom token))
093               ((eq token '/$) (3-read-boolean stream))
094               (t token))))
095
096   (defun 3-read-boolean (stream)
097      (let ((a (readch stream)))
098         (cond ((memq a '(T /t)) '$T)
099               ((memq a '(F /f)) '$F)
100               (t (3-illegal-boolean a)))))
101
102   (defun 3-read-pair (stream)
103      (let ((a (3-read* stream))
104            (b (3-read* stream '(|.| |)|))))
105         (if (eq b '|.|)
106             (prog1 (cons a (3-read* stream))
107                    (setq b (read stream))
108                    (if (not (eq b '|)|)) (3-illegal-char b)))
109             (do ((b b (3-read* stream '(|)|)))
110                  (c nil (cons b c)))
111                 ((eq b '|)|) (list* a '~RAIL~ (nreverse c)))))))
112
113   (defun 3-read-rail (stream)
114      (do ((a nil (cons b a))
115           (b (3-read* stream '(|]|)) (3-read* stream '(|]|))))
116          ((eq b '|]|) (cons '~RAIL~ (nreverse a)))))
117
118   )                                    ; End of eval-when
```

```
(eval-when (eval load compile)          ; Start another eval-when, since the following
                                        ; needs to be read in using 3-READ

;;; BACKQUOTE    3-BACKQ-MACRO and 3-COMMA-MACRO are run on reading: they
;;; ---------    put calls to ~3-BACKQUOTE and ~3-COMMA into the structures
;;;              they build, which are then run on exit.  This allows the
;;;              expansion to happen from the inside out.

(defun 3-backq-macro (stream)
   (let ((3*backquote-depth (1+ 3*backquote-depth)))
      (macroexpand (list '~3-BACKQUOTE (read stream)))))

(defun 3-comma-macro (stream)
   (if (< 3*backquote-depth 1) (3-error '|Unscoped comma|))
   (let ((3*backquote-depth (1- 3*backquote-depth)))
      (cons '~3-COMMA (read stream))))

;;; The second argument to the next 3 procedures is a flag: NIL if the
;;; backquote was at this level; T if not (implying that coalescing can
;;; happen if possible).

(defun 3-expand (x f)
   (caseq (3-type x)
      (PAIR (3-expand-pair x f))
      (RAIL (3-expand-rail x f))
      (T ↑x)))

(defun 3-expand-pair (x f)
   (cond ((eq (car x) '~3-COMMA) (cdr x))       ; Found a ",<expr>".
         ((eq (car x) '~3-BACKQUOTE)            ; Recursive use of backq, so
          (3-expand (macroexpand x) f))         ; expand the inner one and then
         (t (let ((a (3-expand (car x) t))      ; this one.
                  (d (3-expand (cdr x) t)))
              (if (and f (3-handle a) (3-handle d))
                  ↑(cons (cdr a) (cdr d))       ; Do the cons now if possible;
                  `\(PCONS ~,a ~,d))))))        ; else use MACLISP's backquote
                                                ; to form a call to PCONS.

(defun 3-expand-rail (rail f)
  (do ((rail (3-strip rail) (3-strip rail))
       (elements nil (cons (3-expand (car rail) t) elements)))
      ((null rail)
       (if (and f (apply 'and (mapcar '3-handle elements)))
           ↑(cons '~RAIL~ (mapcar 'cdr (nreverse elements)))
           `(RCONS ~RAIL~ ,@(nreverse elements))))))

)                                               ; end of eval-when
```

```
166
167   ;;; 3-PRINT    Print out <exp> in 3-LISP notation using notational sugar if
168   ;;; -------    possible.  No preliminary CR is printed (use TERPRI).  Some
169   ;;;            attempt is made to avoid printing known circular structures
170   ;;;            (like <SIMPLE> and <REFLECT> and obvious circular environments
171   ;;;            of a sort that would be generated by Z).
172
173   (defun 3-print (exp)
174      (caseq (3-type exp)
175         (numeral (princ exp))
176         (boolean (princ exp))
177         (atom    (if (memq exp 3*simple-aliases)
178                      (princ '<simple>)
179                      (prin1 exp)))
180         (handle  (princ '|'|) (3-print !exp))
181         (pair    (cond ((eq exp 3*simple-closure) (princ '<simple>))
182                        ((eq exp 3*reflect-closure) (princ '<reflect>))
183                        (t (princ '|(|)
184                           (3-print (car exp))
185                           (if (3-rail (cdr exp))
186                               (if (3-circular-closure-p exp)
187                                   (progn (princ '| <circular-env>|)
188                                          (3-print-elements (cddr exp) 't))
189                                   (3-print-elements (cdr exp) 't))
190                               (princ '| . |) (3-print (cdr exp)))
191                           (princ '|)|)))))
192         (rail    (princ '|[|)
193                  (3-print-elements exp 'nil)
194                  (princ '|]|))))
195
196   (defun 3-print-elements (list flag)
197      (let ((global (3-strip 3*global-environment)))
198           (do ((list (3-strip list) (3-strip list))
199                (flag flag 't))
200               ((null list))
201               (if (eq list global)
202                   (return (princ '| ... <global>|)))
203               (if flag (princ '| |))
204               (3-print (car list)))))
205
206   (defun 3-prompt (level)
207      (terpri)
208      (princ level)
209      (princ '|> |))
210
211   (defun 3-circular-closure-p (exp)
212      (and (< 0 (3-length (cdr exp)))
213           (3-rail (3r-1st (cdr exp)))
214           (< 0 (3-length (3r-1st (cdr exp))))
215           (let ((env? (3r-1st (3r-1st (cdr exp)))))
216               (and (3-rail env?)
217                    (< 1 (3-length env?))
218                    (3-handle (3r-1st env?))
219                    (3-atom !(3r-1st env?))
220                    (3-handle (3r-2nd env?))
221                    (eq exp !(3r-2nd env?))))))
222
```

```
001
002   ;;; Main Processor:
003   ;;; ***************
004   ;;;
005   ;;;
006   ;;; 3-NORMALISE and 3-REDUCE    The second clause in the following takes care
007   ;;; ------------------------    of numerals, booleans, handles, normal-form
008   ;;;                             function designators (applications in terms of
009   ;;; the functions SIMPLE, MACRO, and REFLECT whose args are in normal form),
010   ;;; and normal-form sequence designators (rails whose elements are all in
011   ;;; normal-form).  Thus all normal-form expressions normalise to themselves,
012   ;;; even those (like rails and function-designators) that are not canonical
013   ;;; designators of their referents.
014
015   (defun 3-normalise (exp env cont)
016      (cond ((3-atom exp) (3-apply cont (3-binding exp env)))
017            ((3-normal exp) (3-apply cont exp))
018            ((3-rail exp) (3-normalise-rail exp env cont))
019            (t (3-reduce (car exp) (cdr exp) env cont))))
020
021   (defun 3-reduce (proc args env cont)
022      (3-normalise* proc env
023         `\(~~C0~ [['proc ~,↑proc] ['args ~,↑args] ['env ~,↑env] ['cont ~,↑cont]]  ; C0
024                  '[proc*]
025                  '(selectq (procedure-type proc*)
026                     [reflect ((simple . !(cdr proc*)) args env cont)]
027                     [simple (normalise args env (make-c1 proc* cont))]))))
028
029   ;;; 3-NORMALISE-RAIL    Normalise (the first element of) a rail.
030   ;;; ----------------
031
032   (defun 3-normalise-rail (rail env cont)
033      (if (null (3-strip rail))
034          (3-apply cont rail)
035          (3-normalise* (3r-1st rail) env
036             `\(~~C2~ [['rail ~,↑rail] ['env ~,↑env] ['cont ~,↑cont]]            ; C2
037                      '[element*]
038                      '(normalise-rail (rest rail) env
039                         (lambda simple [rest*]
040                           (cont (prep element* rest*)))))))))
041
042   ;;; 3-PRIMITIVE-REDUCE-SIMPLE  The way each primitive function is treated is
043   ;;; -------------------------  highly dependent on the way that 3-LISP
044   ;;;                            structures are encoded in MACLISP.
045
046   (defun 3-primitive-reduce-simple (proc args cont)
047    (3-rail-check args)
048    (if (eq proc 'referent)
049        (3-normalise* !(3r-1st args) (3r-2nd args) cont)
050        (3-apply cont
051          (caseq proc
052            (simple   `(,3=simple-closure . ,args))
053            (reflect  `(,3=reflect-closure . ,args))
054            (type     ↑(3-ref-type (3r-1st args)))
055            (ef       (if (eq (3-bool-check (3r-1st args)) '$T)
                             (3r-2nd args) (3r-3rd args)))
056            (pcons    ↑(cons !(3r-1st args) !(3r-2nd args)))
057            (car      ↑(car (3-pair-check !(3r-1st args))))
058            (cdr      ↑(cdr (3-pair-check !(3r-1st args))))
059            (length   (3-length (3r-1st args)))
060            (nth      (3-nth (3r-1st args) (3r-2nd args)))
061            (tail     (3-tail (3r-1st args) (3r-2nd args)))
062            (prep     (3-prep (3r-1st args) (3r-2nd args)))
063            (rcons    (3-rcons (3-rail-check args)))
064            (scons    (3-scons (3-rail-check args)))
065            (rplaca   ↑(rplaca (3-pair-check !(3r-1st args)) !(3r-2nd args)))
066            (rplacd   ↑(rplacd (3-pair-check !(3r-1st args)) !(3r-2nd args)))
067            (rplacn   ↑(3-rplacn (3r-1st args) !(3r-2nd args) !(3r-3rd args)))
068            (rplact   ↑(3-rplact (3r-1st args) !(3r-2nd args) !(3r-3rd args)))
```

```
                                                                    Page 4:1
069             (=          (if (3-equal (3r-1st args) (3r-2nd args)) '$T '$F))
070             (read       ↑(3-read))
071             (print      (3-print !(3r-1st args)) (princ '/ ) '$T)
072             (terpri     (terpri) '$T)
073             (+          (+ (3-num-check (3r-1st args)) (3-num-check (3r-2nd args))))
074             (*          (* (3-num-check (3r-1st args)) (3-num-check (3r-2nd args))))
075             (-          (- (3-num-check (3r-1st args)) (3-num-check (3r-2nd args))))
076             (//         (// (3-num-check (3r-1st args)) (3-num-check (3r-2nd args))))
077             (name       ↑(3r-1st args))
078             (*rebind    (3-rebind !(3r-1st args) (3r-2nd args) (3r-3rd args)))  ; for
079             (level      3-level)                                                ; efficiency
080             (t (3-implementation-error))))))
```

```
001
002   ;;; Continuation Application:
003   ;;; ************************
004   ;;;
005   ;;; 3-APPLY   Called with 3-LISP continuations, has to sort them out and do
006   ;;; -------   the right non-reflected thing with those that are tokens of the
007   ;;;           six types (C0 - C5) that are primitively recognized.  In
008   ;;; addition, redexes in terms of primitive procedures (identified by PRIM)
009   ;;; are recognised.  We assume a continuation of the form
010   ;;; (<simple> . [env [arg] body]), and a standard environment structure.
011
012   (defmacro 3a-env (cont) `(3r-1st (cdr ,cont)))
013   (defmacro 3a-arg (cont) `(3r-2nd (cdr ,cont)))
014   (defmacro 3a-1st (env)  `!(3r-2nd (3r-1st ,env)))
015   (defmacro 3a-2nd (env)  `!(3r-2nd (3r-2nd ,env)))
016   (defmacro 3a-3rd (env)  `!(3r-2nd (3r-3rd ,env)))
017   (defmacro 3a-4th (env)  `!(3r-2nd (3r-4th ,env)))
018
019   (defun 3-apply (cont normal-form)
020     (let ((env (3a-env cont)))
021       (if (memq (car cont) 3=simple-aliases)
022           (funcall (car cont) env cont normal-form)
023           (let ((new-level (3-increment-level)))    ; REFLECT UP!
024              (3-reduce cont ↑`\[~,normal-form]      ; ***********
025                        (car new-level) (cdr new-level))))))
026
027   ;;; C0:  Accept a normalised function designator from a pair.  Dispatch
028   ;;; ---  on the function type: if it is SIMPLE, normalise the args; if
029   ;;;      primitive reflective, go do it; otherwise reflect up explicitly.
030
031   (defun ~C0~ (env cont proc)
032     ignore cont
033     (let ((args (3a-2nd env))
034           (env  (3a-3rd env))
035           (cont (3a-4th env)))
036       (caseq (3-proc-type proc)
037         (simple (3-normalise* args env
038                   `\(~~C1~ [['proc ~,↑proc] ['args ~,↑args]      ; C1
039                             ['env ~,↑env] ['cont ~,↑cont]]
040                            '[args*]
041                            '(cond [(= proc* ↑referent)
042                                    (normalise !(1st args) !(2nd args) cont)]
043                                   [(primitive proc*) (cont ↑(!proc* . !args*))]
044                                   [$T (normalise (body proc*)
045                                                  (bind (pattern proc*) args* (env proc*))
046                                                  cont)])))))
047         (reflect (let ((nlevel (3-increment-level))      ; REFLECT UP!
048                        (proc (cdr proc)))                ; ***********
049                    (3-normalise* !(3r-3rd proc)
050                                  (3-bind !(3r-2nd proc)
051                                          `\[~,↑args ~,env ~,cont]
052                                          (3r-1st proc))
053                                  (cdr nlevel)))))))
```

```
054
055   ;;; C1:  Accept the normalised arguments to a SIMPLE application.  Dispatch
056   ;;; ---  on primitives, and reflect down in case we encounter a call to a
057   ;;;       continuation we ourselves once put together.  Also trap explicit calls
058   ;;;       to NORMALISE and REDUCE, for efficiency.
059
060   (defun ~C1~ (env cont args*)
061      ignore cont
062      (let ((proc (3a-1st env)))
063         (cond ((eq (car proc) '~PRIM~)
064                (3-argument-check args* proc)
065                (3-primitive-reduce-simple (3-primitive-simple-id proc)
066                                           args*
067                                           (3a-4th env)))
068               ((memq (car proc) 3=simple-aliases)
069                (3-drop-level (3a-3rd env) (3a-4th env))          ; REFLECT DOWN
070                (3-apply proc !(3r-1st args*)))                   ; ************
071               ((eq proc 3=normalise-closure)
072                (3-drop-level (3a-3rd env) (3a-4th env))          ; REFLECT DOWN
073                (3-normalise* !(3r-1st args*)                     ; ************
074                              (3r-2nd args*)
075                              (3r-3rd args*)))
076               ((eq proc 3=reduce-closure)
077                (3-drop-level (3a-3rd env) (3a-4th env))          ; REFLECT DOWN
078                (3-reduce !(3r-1st args*)                         ; ************
079                          !(3r-2nd args*)
080                          (3r-3rd args*)
081                          (3r-4th args*)))
082               (t (let ((proc* (cdr proc)))
083                    (3-normalise*
084                          !(3r-3rd proc*)
085                          (3-bind !(3r-2nd proc*) args* (3r-1st proc*))
086                          (3a-4th env)))))))
087
088   ;;; C2:  Accept the normalised first element in a rail fragment.
089   ;;; ---  Normalise the rest.
090
091   (defun ~C2~ (env cont element*)
092      ignore cont
093      (3-normalise-rail
094           (3-tail* 1 (3a-1st env))
095           (3a-2nd env)
096           `\(~~C3~ ~,(nconc `\[['element* ~,telement*]] env)      ; C3
097                    '[rest*]
098                    '(cont (prep element* rest*)))))
099
100   ;;; C3:  Accept the normalised tail of a rail fragment.  Put the first
101   ;;; ---  element on the front.
102
103   (defun ~C3~ (env cont rest*)
104      ignore cont
105      (3-apply (3a-4th env) (nconc `\[~,(3a-1st env)] rest*)))
106
107   ;;; C4:  Accept an expression normalised for the top level of a
108   ;;; ---  READ-NORMALISE-PRINT loop.  Print it out and read another.
109   ;;;
110   ;;; On entry here ENV will be bound to the environment of the C4 closure, CONT
111   ;;; will be bound to the whole C4 closure, and NORMAL-FORM will be bound to a
112   ;;; designator of the result of the NORMALISE at the level below.
113
114   (defun ~C4~ (env cont normal-form)
115      (3-prompt 3=level)
116      (3-print !normal-form)
117      (3-prompt 3=level)
118      (3-drop-level 3=global-environment cont)
119      (3-normalise* (3-read) (3-binding 'env env) 3=id-closure))
```

```
120
121   ;;; C5:  Accept the result of normalising an expression wrapped in an
122   ;;; ---  IN-3-LISP macro.  Return answer to the caller.
123
124   (defun ~C5~ (env cont normal-form)
125      ignore env cont
126      (*throw '3-exit normal-form))
127
128   (defun 3-argument-check (args proc)
129     (let ((pattern !(3r-2nd (cdr proc))))
130       (if (and (3-rail pattern)
131                (not (= (3-length args) (3-length pattern))))
132           (3-error '|Wrong number of arguments to a primitive: |
133                    `\(~,(car !(3r-3rd proc)) . ~,args)))))
134
```

```
001
002   ;;; Environments:
003   ;;; =============
004   ;;;
005   ;;; 3-BINDING   Look up a binding in a 3-LISP standard environment
006   ;;; ---------   designator, but, for efficiency, bypass rail type-checking.
007
008   (defun 3-binding (var env)
009      (3-atom-check var)
010      (3-rail-check env)
011      (do ((env (3-strip env) (3-strip env)))
012          ((null env) (3-error `(,var unbound variable -- BINDING)))
013        (if (eq var !(3r-1st (car env))) (return !(3r-2nd (car env))))))
014
015   ;;; 3-BIND   Bind variable structure to argument structure.  Destructures on
016   ;;; ------   rails and sequences.  For efficiency, does rail manipulation by
017   ;;;          itself, saving time and cons'es.  The DO constructs a reversed
018   ;;;          MACLISP rail designator, NREVERSEd on exit.
019
020   (defun 3-bind* (pattern vals)
021     (caseq (3-type pattern)
022       (atom `(\[~,↑pattern ~,↑vals]))
023       (rail (caseq (3-type vals)
024               (rail (do ((binds nil (nconc (3-bind* (car pattern) (car vals)) binds))
025                          (pattern (3-strip pattern) (3-strip pattern))
026                          (vals (3-strip vals) (3-strip vals)))
027                         ((or (null pattern) (null vals))
028                          (cond ((and (null pattern) (null vals))
029                                 (nreverse binds))
030                                ((null vals) (3-error '|Too few arguments supplied|))
031                                (t (3-error '|Too many arguments supplied|))))))
032               (handle (if (3-rail ivals)
033                           (do ((binds nil (nconc (3-bind* (car pattern) ↑(car vals))
034                                                  binds))
035                                (pattern (3-strip pattern) (3-strip pattern))
036                                (vals (3-strip ivals) (3-strip vals)))
037                               ((or (null pattern) (null vals))
038                                (cond ((and (null pattern) (null vals))
039                                       (nreverse binds))
040                                      ((null vals) (3-error '|Too few arguments supplied|))
041                                      (t (3-error '|Too many arguments supplied|))))))
042                           (3-type-error vals '|ATOM, RAIL, or RAIL DESIGNATOR|)))
043               (t (3-type-error vals '|ATOM, RAIL, OR RAIL DESIGNATOR|)))))
044       (t (3-type-error pattern '|ATOM, RAIL, OR RAIL DESIGNATOR|))))
045
046   (defun 3-rebind (var binding env)
047      (3-atom-check var)
048      (3-rail-check env)
049      (if (not (3-normal binding))
050          (3-error '(binding not in normal form -- REBIND/:) binding))
051      (do ((env (3-strip* env) (3-strip* (cdr env))))
052          ((null (cdr env)) (nconc env `\[[~,↑var ~,binding]]))
053        (if (eq var !(3r-1st (cadr env)))
054            (return (3-rplacn 2 (cadr env) binding))))
055      binding)
056
```

```
001
002   ;;; Reflective state management:
003   ;;; ****************************
004   ;;;
005   ;;; 3-STATES is a queue of the environment and continuation of each reflective
006   ;;; level ABOVE the current one (the value of 3-LEVEL), if they were ever
007   ;;; explicitly generated (all relevant ones BELOW the current level are of
008   ;;; course being passed around explicitly in 3-LISP programs).
009
010   (defun 3-drop-level (env cont)
011      (push (cons env cont) 3-states)
012      (setq 3-level (1- 3-level)))
013
014   (defun 3-increment-level ()
015      (setq 3-level (1+ 3-level))
016      (if (not (null 3-states))
017          (pop 3-states)
018          (cons 3-global-environment
019                `\(~~C4~ ~,(nconc `\[['env ~,↑3-global-environment]]
020                                  3-global-environment)
021                          '[normal-form]
022                          '(block (prompt (level))
023                                  (print normal-form)
024                                  (read-normalise-print env))))))
025
```

```
001
002   ;;; Rail Management:
003   ;;; ================
004   ;;;
005   ;;;
006   ;;; 3-RCONS   Make a new rail (or sequence designator) out of the args
007   ;;; 3-SCONS
008   ;;; -------
009
010   (defun 3-rcons (args)
011     (do ((args (3-strip (3-rail-check args)) (3-strip args))
012          (new nil (cons !(car args) new)))
013         ((null args) ↑(cons '~RAIL~ (nreverse new)))))
014
015   (defun 3-scons (args)
016     (do ((args (3-strip (3-rail-check args)) (3-strip args))
017          (new nil (cons (car args) new)))
018         ((null args) (cons '~RAIL~ (nreverse new)))))
019
020   ;;; 3-RS  Macro that takes two forms, one for rails and one for sequences,
021   ;;; ----  and wraps the appropriate type dispatch around them.
022
023   (defmacro 3-rs (exp rail-form seq-form)
024      `(caseq (3-type ,exp)
025          (handle ,rail-form)
026          (rail ,seq-form)
027          (t (3-ref-type-error ,exp '|RAIL OR SEQUENCE|))))
028
029   ;;; 3-PREP       -- These four kinds are defined over both rails and sequences.
030   ;;; 3-LENGTH        They are all defined in terms of *-versions, which operate
031   ;;; 3-TAIL          on the implementing rails.
032   ;;; 3-NTH
033
034   (defun 3-prep (el exp)
035      (3-rs exp ↑(list* '~RAIL~ !el (3-rail-check !exp))
036                (list* '~RAIL~ el exp)))
037
038   (defun 3-length (exp)
039      (3-rs exp (3-length* (3-rail-check !exp))
040                (3-length* exp)))
041
042   (defun 3-tail (n exp)
043      (3-rs exp ↑(3-tail* n (3-rail-check !exp))
044                (3-tail* n exp)))
045
046   (defun 3-nth (n exp)
047      (3-rs exp ↑(car (3-nthcdr* n (3-rail-check !exp)))
048                (car (3-nthcdr* n exp))))
049
050   ;;; 3-RPLACN   Defined only on RAILS.
051   ;;; --------
052
053   (defun 3-rplacn (n rail el)
054      (rplaca (3-nthcdr* n (3-rail-check rail)) el)
055      rail)
056
057   (defun 3-nthcdr* (n rail)
058      (if (< n 1) (3-index-error n rail))
059      (do ((i 1 (1+ i))
060           (rest (3-strip rail) (3-strip rest)))
061          ((or (= n i) (null rest))
062           (if (null rest)
063               (3-index-error n rail)
064               rest))))
```

```
065
066  (defun 3-tail* (n o-rail)
067    (if (< n 0) (3-index-error n o-rail))
068    (if (zerop n)
069        o-rail
070        (do ((i 0 (1+ i))
071             (rail (3-strip* o-rail) (3-strip* (cdr rail))))
072            ((or (= n i) (null (cdr rail)))
073             (if (= n i)
074                 (if (eq (car rail) '~RAIL~)
075                     rail
076                     (let ((tail (cons '~RAIL~ (cdr rail))))
077                       (rplacd rail tail) ; Splice in a new header
078                       tail))
079                 (3-error `(,n is too large for a tail of) o-rail))))))
080
081  ;;; RPLACT is what all the trouble is about.  A tempting implementation is:
082  ;;;
083  ;;;   (defmacro 3-rplact (n r1 r2) `(cdr (rplacd (3-tail ,n ,r1) ,r2)))
084  ;;;
085  ;;; but this has two problems.  First, it can generate an unnecessary header,
086  ;;; since 3-TAIL may construct one, even though r2 is guaranteed to have one
087  ;;; already.  Second, some uses of this (such as (RPLACT 1 X X)) would generate
088  ;;; circular structures.  The following version avoids these problems:
089
090  (defun 3-rplact (n r1 r2)
091     (3-rail-check r1)
092     (3-rail-check r2)
093     (if (< n 0) (3-index-error n r1))
094     (do ((i 0 (1+ i))
095          (last r1 rail)
096          (rail (3-strip* r1) (3-strip* (cdr rail))))
097         ((or (= n i) (null (cdr rail)))
098          (progn
099           (if (not (= n i)) (3-index-error n r1))
100           (if (let ((r2-headers (do ((r2 r2 (cdr r2))
101                                      (heads nil (cons r2 heads)))
102                                     ((not (eq (car r2) '~RAIL~)) heads))))
103                 (do ((r1-header (cdr last) (cdr r1-header)))
104                     ((not (eq (car r1-header) '~RAIL~)) 't)
105                   (if (memq r1-header r2-headers) (return 'nil))))
106               (rplacd rail r2))
107           r1))))
108
```

```
001
002   ;;; Typing and Type Checking:
003   ;;; ************************
004
005   (eval-when (load eval compile)                  ; Backquote needs this
006
007   (defun 3-type (exp)
008       (cond ((fixp exp) 'numeral)
009             ((memq exp '($T $F)) 'boolean)
010             ((symbolp exp) 'atom)
011             ((eq (car exp) '~RAIL~) 'rail)
012             ((eq (car exp) '~QUOTE~) 'handle)
013             (t 'pair)))
014
015   )                                               ; end of eval-when
016
017   ;;; 3-boolean and 3-numeral are macros, defined above.
018
019   (defun 3-atom (e) (and (symbolp e) (not (memq e '($T $F)))))
020   (defun 3-rail (e) (and (list? e) (eq (car e) '~RAIL~)))
021   (defun 3-pair (e) (eq (3-type e) 'pair))
022
023   (eval-when (load eval compile)
024   (defun 3-handle (e)  (and (list? e) (eq (car e) '~QUOTE~)))
025   )
026
027   (defun 3-atom-check   (e) (if (3-atom e)    e (3-type-error e 'atom)))
028   (defun 3-rail-check   (e) (if (3-rail e)    e (3-type-error e 'rail)))
029   (defun 3-pair-check   (e) (if (3-pair e)    e (3-type-error e 'pair)))
030   (defun 3-handle-check (e) (if (3-handle e)  e (3-type-error e 'handle)))
031   (defun 3-num-check    (e) (if (3-numeral e) e (3-type-error e 'numeral)))
032   (defun 3-bool-check   (e) (if (3-boolean e) e (3-type-error e 'boolean)))
033
034   ;;; 3-REF-TYPE   Returns the type of the entity designated by the 3-LISP
035   ;;; ----------   object encoded as the argument.
036
037   (defun 3-ref-type (exp)
038     (caseq (3-type exp)
039       (numeral 'number)
040       (boolean 'truth-value)
041       (rail    'sequence)
042       (handle  (3-type (cdr exp)))
043       (pair    (if (or (eq (car exp) 3=simple-closure)
044                        (eq (car exp) 3=reflect-closure)
045                        (memq (car exp) 3=simple-aliases))
046                   'function
047                   (3-error '(not in normal form -- REF-TYPE/:) exp)))
048       (atom    (3-error '(not in normal form -- REF-TYPE/:) exp))))
049
050   ;;; 3-REF   Returns the referent of the argument, which must either be a
051   ;;; -----   handle or a rail of handles, since the only kinds of ref's we
052   ;;;         can return are s-expressions.
053
054   (defun 3-ref (exp)
055     (cond ((3-handle exp) (cdr exp))
056           ((3-rail exp)
057            (do ((rail (3-strip exp) (3-strip rail))
058                 (elements nil (cons !(car rail) elements)))
059                ((null rail) (cons '~RAIL~ (nreverse elements)))
060              (if (not (3-handle (car rail)))
061                  (3-ref-type-error exp '|SEQUENCE OF S-EXPRESSIONS|)))))
062           (t (3-ref-type-error exp '|S-EXPRESSION OR SEQUENCE OF S-EXPRESSIONS|))))
063
064   ;;; 3-PROC-TYPE   Returns the procedure type of the argument
065   ;;; -----------
066
067   (defun 3-proc-type (proc)
068     (3-pair-check proc)
069     (cond ((eq (car proc) 3=simple-closure) 'simple)
070           ((memq (car proc) 3=simple-aliases) 'simple)
071           ((eq (car proc) 3=reflect-closure) 'reflect)
072           (t (3-type-error proc 'closure))))
```

```
001
002   ;;; Identity and Normal-form Predicates:
003   ;;; ************************************
004   ;;;
005   ;;;
006   ;;; 3-CANONICALISE   Maps aliases onto their proper identity.
007   ;;; --------------
008
009   (defun 3-canonicalise (exp)
010      (if (and (symbolp exp) (memq exp 3*simple-aliases))
011          3*simple-closure
012          exp))
013
014   ;;; 3-EQUAL   True just in case arguments implement the same 3-LISP object.
015   ;;; -------
016
017   (defun 3-equal (e1 e2)
018     (and (eq (3-type e1) (3-type e2))
019          (caseq (3-type e1)
020            (handle (let ((r1 (3-canonicalise !e1))
021                          (r2 (3-canonicalise !e2)))
022                      (or (eq r1 r2)
023                          (and (3-rail r1)
024                               (3-rail r2)
025                               (eq (3-strip* r1) (3-strip* r2)))
026                          (and (3-handle r1)
027                               (3-handle r2)
028                               (3-equal r1 r2)))))
029            (boolean (eq e1 e2))
030            (numeral (= e1 e2))
031            (rail (do ((e1 (3-strip e1) (3-strip e1))
032                       (e2 (3-strip e2) (3-strip e2)))
033                      ((null e1) (null e2))
034                    (if (not (3-equal (car e1) (car e2)))
035                        (return 'nil))))
036            (t (3-error '|= is defined only over s-expressions,
037                          numerals, truth-values, and some sequences|))))))
038
039   ;;; 3-NORMAL   True in case argument is in normal form.
040   ;;; --------
041
042   (defun 3-normal (exp)
043      (or (3-handle exp) (3-pnormal exp)))
044
045   (defun 3-pnormal (exp)
046     (or (fixp exp)
047         (memq exp '($T $F))
048         (and (list? exp)
049              (or (eq (car exp) 3*simple-closure)
050                  (eq (car exp) 3*reflect-closure)
051                  (memq (car exp) 3*simple-aliases))
052              (3-rail (cdr exp))
053              (3-normal (3r-1st (cdr exp)))
054              (3-normal (3r-2nd (cdr exp)))
055              (3-normal (3r-3rd (cdr exp))))
056         (and (3-rail exp)
057              (do ((exp (3-strip exp) (3-strip exp)))
058                  ((null exp) 't)
059                (if (not (3-normal (car exp))) (return 'nil)))))))
060
```

```
001
002   ;;; Top Level:
003   ;;; ==========
004
005   (defmacro loop-catch (tag &rest body)
006      `(do nil (nil) (*catch ,tag ,@body)))
007
008   ;;; 3-LOGIN     Used only for obscure reasons on the LISP machine, having
009   ;;; 3-LOGOUT    to do with compatibility with other users, recovery from
010   ;;; --------    warm boots, and so forth.
011
012   (defun 3-logout ()
013     (setf (tv:io-buffer-input-function tv:kbd-io-buffer)
014           nil)
015     (setq readtable S-readtable))
016
017   (defun 3-login ()
018     (or (boundp '3-global-environment) (3-init))
019     (setq base 10. ibase 10. *nopoint t)
020     (setq readtable L-readtable))
021
022   ;;; 3-LISP    Starts up the 3-LISP processor.  The 3-LEVEL-LOOP loop is
023   ;;; ------    only run on initialisation and errors; otherwise the
024   ;;;           READ-NORMALISE-PRINT loop is run out of ~C4~.
025
026   (defun 3-lisp ()
027     (setf (tv:io-buffer-input-function tv:kbd-io-buffer)
028           (let-closed ((3-process current-process))
029             #'3-interrupt-handler))
030     (or (boundp '3-global-environment) (3-init))
031     (*catch '3-exit
032      (loop-catch '3-top-loop
033       (let ((3-in-use t))
034          (setq 3-level 0
035                3-states nil)
036          (loop-catch '3-level-loop
037             (3-prompt (1+ 3-level))
038             (setq 3-a1 (3-read)
039                   3-a2 3-global-environment
040                   3-a3 3-id-closure)
041             (loop-catch '3-main-loop (3-normalise 3-a1 3-a2 3-a3)))))))
042
043   ;;; 3-LISPIFY   Normalises its argument (should be a 3-LISP expression)
044   ;;; ---------   at the top level of the level 1 3-LISP environment (intended
045   ;;;             for use by IN-3-LISP).
046
047   (defun 3-lispify (expr)
048     (setq 3-level 1
049           3-states nil
050           3-a1 expr
051           3-a2 3-global-environment
052           3-a3 `\(~~C5~ ~,3-a2 []))
053     (*catch '3-exit
054        (loop-catch '3-main-loop (3-normalise 3-a1 3-a2 3-a3))))
055
```

```
001
002   ;;; Errors and Interrupts:
003   ;;; **********************
004
005   ;;; 3-ERROR   General error handler.  MESSAGE is to be printed by MACLISP's
006   ;;; -------   PRINC, whereas EXPR is printed by 3-PRINT.
007
008   (defun 3-error (message &optional expr (label '|ERROR: |))
009       (terpri)
010       (princ label)
011       (if (atom message)
012           (princ message)
013           (mapc #'(lambda (el) (princ el) (princ '| |))
014                 message))
015       (if expr (3-print expr))
016       (break 3-bkpt 3*break-flag)
017       (if 3*in-use
018           (*throw '3-level-loop nil)
019           (3-lisp)))
020
021   ;;; 3-TYPE-ERROR                          3-ILLEGAL-CHAR
022   ;;; 3-INDEX-ERROR                         3-ILLEGAL-ATOM
023   ;;; 3-IMPLEMENTATION-ERROR                3-ILLEGAL-BOOLEAN
024   ;;; ----------------------                -----------------
025
026   (defun 3-type-error (exp type)
027      (3-error `(expected a ,(implode `(,@(explodec type) #/,))
028                            but found the ,(3-type exp))
029               exp '|TYPE-ERROR: |))
030
031   (defun 3-ref-type-error (exp type)
032     (3-error `(expected a ,(implode `(,@(explodec type) #/,))
033                           but found the ,(3-ref-type exp))
034              exp '|TYPE-ERROR: |))
035
036   (defun 3-index-error (n rail)
037     (3-error `(,n is out of range for) rail '|INDEX-ERROR: |))
038
039   (defun 3-implementation-error () (3-error '|Illegal implementation state!|))
040
041   (defun 3-illegal-char (char)
042      (3-error `(unexpected ,(implode `(|"| ,@(explodec char) |"|)))
043               nil '|NOTATION-ERROR: |))
044
045   (defun 3-illegal-boolean (exp)
046      (3-error `(expected a boolean/, but found ,(implode `($ ,@(explodec exp))))
047               nil '|NOTATION-ERROR: |))
048
049   (defun 3-illegal-atom (atom)
050      (3-error `(The atom ,atom is reserved in this implementation)
051               nil '|STRUCTURE-ERROR: |))
```

```
052
053   ;;; INTERRU?TS (this code is LISP machine specific):
054   ;;; -----------------------------------------------
055
056   (defun 3-interrupt-handler (ignore character)
057     (values character
058             (and tv:selected-window
059                  (eq 3=process (funcall tv:selected-window ':PROCESS))
060                  (boundp '3=in-use)
061                  (selectq character
062                    (#↑B/G
063                       (setq si:inhibit-scheduling-flag nil)
064                       (process-run-temporary-function
065                         "3=Main-Quit" 3=process ':INTERRUPT
066                         #'(lambda () (3-quit-interrupt '3-top-loop)))
067                       T)
068                    (#↑B/F
069                       (setq si:inhibit-scheduling-flag nil)
070                       (process-run-temporary-function
071                         "3=Level-Quit" 3=process ':INTERRUPT
072                         #'(lambda () (3-quit-interrupt '3-level-loop)))
073                       T)
074                    (#↑B/E
075                       (setq si:inhibit-scheduling-flag nil)
076                       (process-run-temporary-function
077                         "3=Flip" 3=process ':INTERRUPT
078                         #'(lambda ()
079                             (if 3=in-use
080                                 (progn (princ '|To LISP!|) (terpri)
081                                        (*throw '3-exit (ascii 0)))
082                                 (progn (princ '| To 3-LISP!|)
083                                        (3-lisp)))))
084                       T)))))
085
086   (defun 3-quit-interrupt (tag)
087      (if 3=in-use
088          (progn (princ '| QUIT!|)
089                 (terpri)
090                 (*throw tag nil))
091          (*throw 'sys:command-level nil)))
092
093
```

```
001
002   ;;; Initialisation:
003   ;;; ***************
004
005   (defun 3-init ()
006    (princ '|   (initialising 3-LISP reflective model -- this takes a few minutes)|)
007    (setq
008      3=in-use nil
009      3=level 1
010      3=break-flag t
011      3=simple-aliases '(~C0~ ~C1~ ~C2~ ~C3~ ~C4~ ~C6~ ~PRIM~)
012      3=normalise-closure nil                ; These will be set to real values
013      3=reduce-closure nil                   ; later, but will be referenced first
014      3=id-closure nil
015      3=global-environment (3-initial-environment)
016      prinlength 6                           ; In case environments
017      prinlevel 4                            ; are printed by LISP
018      base 10.                               ; Since 3-LISP assumes base 10 integers
019      ibase 10.                              ; and we use the straight LISP printer
020      *nopoint T)
021     (setq 3=simple-closure (3-binding 'simple 3=global-environment)
022           3=reflect-closure (3-binding 'reflect 3=global-environment))
023     (3-define-utilities-0)
024     (3-define-reflective)
025     (setq 3=normalise-closure (3-binding 'normalise 3=global-environment)
026           3=reduce-closure (3-binding 'reduce 3=global-environment))
027     (3-define-utilities-1)                 ; The order here is crucial: have to
028     (setq 3=id-closure (3-binding 'id 3=global-environment))
029     (3-define-utilities-2)                 ; get the def's marked before these.
030     (3-define-utilities-3))
031
032   ;;; 3-INITIAL-ENVIRONMENT    Returns a new initialised 3-LISP environment,
033   ;;; ---------------------    with each of the names of primitive functions
034   ;;;                          bound to a circular definition, closed in the new
035   ;;; environment, that betrays both the type and the number of arguments.  For
036   ;;; example, CAR is bound to the normalisation of (LAMBDA SIMPLE [X] (CAR X)).
037   ;;; This could just be a constant list that was copied, but is instead
038   ;;; generated by the following function, that fakes the normalisation process
039   ;;; and then side-effects the result to make the environment structures
040   ;;; circular.
041
042   (defun 3-initial-environment ()
043      (let ((env `\[['global ~~~~]
044                    ~,@(mapcar '3-make-primitive-closure
045                               (3-circular-closures))]))
046        (mapcar #'(lambda (entry)
047                    (3-rplacn 1 (cdr !(3r-2nd entry)) env))
048                (cddr env))
049        (3-rplacn 2 (3r-1st env) tenv)
050        env))
```

```
051
052   ;;; 3-MAKE-PRIMITIVE-CLOSURE   Constructs the primitive definitions.
053   ;;; ------------------------
054
055   (defun 3-make-primitive-closure (entry)
056     (let ((name (car entry))
057           (def (cdadr entry)))
058       `\[~,↑name ~,↑(cons '~PRIM~ `\[~~dummy~ ~,↑(3r-2nd def) ~,↑(3r-3rd def)])]))
059
060   (defun 3-circular-closures ()
061     '((terpri   \(lambda simple [] (terpri)))
062       (read     \(lambda simple [] (read)))
063       (type     \(lambda simple [exp] (type exp)))
064       (car      \(lambda simple [pair] (car pair)))
065       (cdr      \(lambda simple [pair] (cdr pair)))
066       (length   \(lambda simple [vector] (length vector)))
067       (print    \(lambda simple [exp] (print exp)))
068       (name     \(lambda simple [exp] (name exp)))
069       (=        \(lambda simple [a b] (= a b)))
070       (pcons    \(lambda simple [a b] (pcons a b)))
071       (rcons    \(lambda simple args (rcons . args)))
072       (scons    \(lambda simple args (scons . args)))
073       (prep     \(lambda simple [element vector] (prep element vector)))
074       (nth      \(lambda simple [n vector] (nth n vector)))
075       (tail     \(lambda simple [n vector] (tail n vector)))
076       (rplaca   \(lambda simple [a pair] (rplaca a pair)))
077       (rplacd   \(lambda simple [d pair] (rplacd d pair)))
078       (rplacn   \(lambda simple [n rail element] (rplacn n rail element)))
079       (rplact   \(lambda simple [n rail tail] (rplact n rail tail)))
080       (+        \(lambda simple [a b] (+ a b)))
081       (-        \(lambda simple [a b] (- a b)))
082       (*        \(lambda simple [a b] (* a b)))
083       (//       \(lambda simple [a b] (/ a b)))
084       (referent \(lambda simple [exp env] (referent exp env)))
085       (simple   \(lambda simple [env pattern body] (simple env pattern body)))
086       (reflect  \(lambda simple [env pattern body] (reflect env pattern body)))
087       (ef       \(lambda simple [premise c1 c2] (ef premise c1 c2)))
088       (*rebind  \(lambda simple [var binding env] (*rebind var binding env)))
089       (level    \(lambda simple [] (level)))))
090
```

```
001
002   ;;; 3-LISP:  Reflective Processor:
003   ;;; ****************************
004
005   (defun 3-define-reflective ()
006    (in-3-lisp \[
007
008   (define READ-NORMALISE-PRINT
009     (lambda simple [env]
010        (block (prompt (level))
011               (let [[normal-form (normalise (read) env id)]]
012                 (prompt level)
013                 (print normal-form)
014                 (read-normalise-print env)))))
015
016   (define NORMALISE
017     (lambda simple [exp env cont]
018        (cond [(normal exp) (cont exp)]
019              [(atom exp) (cont (binding exp env))]
020              [(rail exp) (normalise-rail exp env cont)]
021              [(pair exp) (reduce (car exp) (cdr exp) env cont)])))
022
023   (define REDUCE
024     (lambda simple [proc args env cont]
025        (normalise proc env
026           (lambda simple [proc*]                                          ; C0
027              (selectq (procedure-type proc*)
028                  [reflect ((simple . !(cdr proc*)) args env cont)]
029                  [simple (normalise args env (make-c1 proc* cont))])))))
030
031   (define MAKE-C1
032     (lambda simple [proc* cont]
033        (lambda simple [args*]                                           ; C1
034           (cond [(= proc* !referent)
035                  (normalise !(1st args) !(2nd args) cont)]
036                 [(primitive proc*) (cont !(!proc* . !args*))]
037                 [$T (normalise (body proc*)
038                                (bind (pattern proc*) args* (env proc*))
039                                cont)]))))
040
041   (define NORMALISE-RAIL
042     (lambda simple [rail env cont]
043        (if (empty rail)
044            (cont `[])
045            (normalise (1st rail) env
046                (lambda simple [element*]                                ; C2
047                   (normalise-rail (rest rail) env
048                      (lambda simple [rest*]                             ; C3
049                         (cont (prep element* rest*)))))))))
050
051   ]))
052
```

```
001
002   ;;; 3-LISP:  Utility Support:
003   ;;; ========================
004
005   ;;; 3-DEFINE-UTILITIES-0 sets up the definitions of SET, DEFINE, LAMBDA,
006   ;;; and Z, so that subsequent defining can proceed regularly.  The technique
007   ;;; is to bootstrap our way up through temporary versions of a bunch of
008   ;;; procedures, so as to put ourselves into a position where more adequate
009   ;;; versions can be manageable defined.
010
011   (defun 3-define-utilities-0 ()
012     (in-3-lisp \[
013
014   ;;; First define CURRENT-ENV (so that down-arrow can work) and LAMBDA:
015
016   (rplact (length global)
017           †global
018           `[['CURRENT-ENV ,††(reflect [['name †name]]
019                                       '[[] env cont]
020                                       '(cont †env))]
021             ['LAMBDA ,††(reflect ((reflect [['name †name]]
022                                            '[[] env cont]
023                                            '(cont †env)))
024                                  '[[type pattern body] env cont]
025                                  '(cont †![pcons type †[env pattern body]]))]]]
026
027   ;;; Next tentative versions of SET, and a real version of Z (though we can't
028   ;;; use LET or BLOCK in defining Z, this definition is equivalent to the one
029   ;;; given in the text).  In the following definition of &SET, *REBIND is used,
030   ;;; rather than &REBIND, for efficiency (*REBIND is provided primitively).  We
031   ;;; have left in the full definition of &REBIND, to show how it would go: it
032   ;;; is merely unacceptably slow.
033
034   (rplact (length global)
035           †global
036           `[['&SET ,††(lambda reflect [[var binding] env cont]
037                         (cont (*rebind var †[binding env])))]
038             ['Z ,††(lambda simple [fun]
039                       ((lambda simple [temp]
040                          ((lambda simple [closure]
041                             ((lambda simple [? ?] temp)
042                              (rplaca †temp (car closure))
043                              (rplacd †temp (cdr closure))))
044                           †(fun temp)))
045                        (lambda simple args (error 'partial-closure-used))))]]]
046
047   ;;; Now a temporary version of REBIND (which is recursive, and uses an explicit
048   ;;; call to Z in its construction), and a temporary DEFINE that doesn't protect Z,
049   ;;; and that expands the macro explicitly:
050
051   (rplact (length global)
052           †global
053           `[['&REBIND
054             ,††(Z (lambda simple [&rebind]
055                     (lambda simple [var binding env]
056                       ((ef (= (length env) 0)
057                            (lambda simple []
058                              (rplact 0 †env †[[var binding]]))
059                            (lambda simple []
060                              ((ef (= var (nth 1 (nth 1 env)))
061                                   (lambda simple []
062                                     (rplacn 2 †(nth 1 env) †binding))
063                                   (lambda simple []
064                                     (&rebind var binding (tail 1 env))))))))))]
065             ['DEFINE ,††(lambda reflect[[label form] env cont]
066                           ((lambda simple ? (cont label))
067                            †(referent `(&set ,label
068                                              (z (lambda simple [,label] ,form)))
069                                       env)))]]]
070   ]))
```

```
071
072   ;;; In general there is a sense of order here: IF, for example, must preceed
073   ;;; LET; hence it cannot use LET in its own definition.  And so on and so forth;
074   ;;; it takes a little care to build things up in a consistent and non-circular
075   ;;; manner.
076
077   (defun 3-define-utilities-1 ()
078     (in-3-lisp \[
079
080   (define ID (lambda simple [x] x))
081
082   (define 1ST (lambda simple [x] (nth 1 x)))
083   (define 2ND (lambda simple [x] (nth 2 x)))
084   (define 3RD (lambda simple [x] (nth 3 x)))
085   (define 4TH (lambda simple [x] (nth 4 x)))
086
087   (define REST (lambda simple [x] (tail 1 x)))
088   (define FOOT (lambda simple [x] (tail (length x) x)))
089
090   (define EMPTY  (lambda simple [x] (= (length x) 0)))
091   (define UNIT   (lambda simple [x] (= (length x) 1)))
092   (define DOUBLE (lambda simple [x] (= (length x) 2)))
093
094   (define ATOM        (lambda simple [x] (= (type x) 'atom)))
095   (define RAIL        (lambda simple [x] (= (type x) 'rail)))
096   (define PAIR        (lambda simple [x] (= (type x) 'pair)))
097   (define NUMERAL     (lambda simple [x] (= (type x) 'numeral)))
098   (define HANDLE      (lambda simple [x] (= (type x) 'handle)))
099   (define BOOLEAN     (lambda simple [x] (= (type x) 'boolean)))
100
101   (define NUMBER      (lambda simple [x] (= (type x) 'number)))
102   (define SEQUENCE    (lambda simple [x] (= (type x) 'sequence)))
103   (define TRUTH-VALUE (lambda simple [x] (= (type x) 'truth-value)))
104
105   (define FUNCTION    (lambda simple [x] (= (type x) 'function)))
106
107   (define PRIMITIVE
108     (lambda simple [proc]
109        (member proc
110                ↑[type = pcons car cdr rcons scons prep length nth tail rplaca
111                  rplacd rplacn rplact simple reflect ef name referent + * - /
112                  read print])))
113
114   (define PROMPT (lambda simple [] (block (print ↑(level)) (print '>))))
115
116   (define BINDING
117     (lambda simple [var env]
118        (cond [(empty env) (error 'unbound-variable)]
119              [(= var (1st (1st env))) (2nd (1st env))]
120              [$t (binding var (rest env))])))
121
122   (define ENV     (lambda simple [proc] !(1st (cdr proc))))
123   (define PATTERN (lambda simple [proc] !(2nd (cdr proc))))
124   (define BODY    (lambda simple [proc] !(3rd (cdr proc))))
125
126   (define PROCEDURE-TYPE
127     (lambda simple [proc]
128        (select (car proc)
129          [↑simple 'simple]
130          [↑reflect 'reflect])))
```

```
131
132   (define XCONS
133     (lambda simple args
134       (pcons (1st args) (rcons . (rest args)))))
135
136   (define BIND
137     (lambda simple [pattern args env]
138       !(join ↑(match pattern args) ↑env)))
139
140   (define MATCH
141     (lambda simple [pattern args]
142       (cond [(atom pattern) [[pattern args]]]
143             [(handle args) (match pattern (map name !args))]
144             [(and (empty pattern) (empty args)) (scons)]
145             [(empty pattern) (error 'too-many-arguments)]
146             [(empty args) (error 'too-few-arguments)]
147             [$T !(join ↑(match (1st pattern) (1st args))
148                        ↑(match (rest pattern) (rest args)))])))
149
150   (define IF
151     (lambda reflect [args env cont]
152       ((ef (rail args)
153            (lambda simple []
154              ((lambda simple [premise c1 c2]
155                  (normalise premise env
156                    (lambda simple [premise*]
157                      ((ef (= premise* '$T)
158                           (lambda simple [] (normalise c1 env cont))
159                           (lambda simple [] (normalise c2 env cont)))))))
160                . args))
161            (lambda simple []
162              (normalise args env
163                (lambda simple [[premise c1 c2]]
164                  (cont (ef (= premise '$T) c1 c2)))))))))
165
166   (define MEMBER
167     (lambda simple [element vector]
168       (cond [(empty vector) $F]
169             [(= element (1st vector)) $T]
170             [$t (member element (rest vector))])))
171
172   (define PREP*
173     (lambda simple args
174       (cond [(empty args) (error 'too-few-args)]
175             [(unit args) (1st args)]
176             [(double args) (prep . args)]
177             [$T (prep (1st args) (prep* . (rest args)))])))
178
179   (define NORMAL
180     (lambda simple [x]
181       (selectq (type x)
182         [numeral $T]
183         [boolean $T]
184         [handle  $T]
185         [atom    $F]
186         [rail (and . (map normal x))]
187         [pair (and (member (car pair) ↑[simple reflect])
188                    (normal (cdr pair)))])))
189
190   (define NOT (lambda simple [x] (if x $F $T)))
```

```
191
192   (define COPY
193     (lambda simple [rail]
194       (if (empty rail)
195           (rcons)
196           (prep (1st rail) (copy (rest rail))))))
197
198   (define JOIN
199     (lambda simple [rail1 rail2]
200       (rplact (length rail1) rail1 rail2)))
201
202   (define APPEND
203     (lambda simple [rail1 rail2]
204       (join (copy . rail1) rail2)))
205
206   (define REDIRECT
207     (lambda simple [index rail new-tail]
208       (if (< index 1)
209           (error 'redirect-called-with-too-small-an-index)
210           (rplact (- index 1)
211                   rail
212                   (prep (nth index rail) new-tail)))))
213
214   (define PUSH
215     (lambda simple [element stack]
216       (rplact 0 stack
217               (prep element
218                     (if (empty stack)
219                         `[]
220                         (prep (1st stack) (rest stack)))))))
221
222   (define POP
223     (lambda simple [stack]
224       (if (empty stack)
225           (error 'stack-underflow)
226           (block1 (1st stack)
227                   (rplact 0 stack (rest stack))))))
228
229   (define MACRO
230     (lambda simple [def-env pattern body]
231       (reflect def-env
232                `[,pattern env cont]
233                `(normalise ,body env cont))))
234
235   (define SHACRO
236     (lambda simple [def-env pattern body]
237       (reflect def-env
238                '[args env cont]
239                `(normalise args env
240                    (lambda simple [,pattern]
241                       (normalise ,body env cont))))))
242
243   ]))
```

```
244
245   (defun 3-define-utilities-2 ()
246     (in-3-lisp \[
247
248   (define LET
249     (lambda macro [list body]
250       `((lambda simple ,(map 1st list) ,body)
251         .,(map 2nd list))))
252
253   (define LET*
254     (lambda macro [list body]
255       (if (empty list)
256           body
257           `((lambda simple ,(1st (1st list))
258                ,(let* (rest list) body))
259             .,(2nd (1st list))))))
260
261   (define SELECTQ
262     (lambda macro args
263       `(let [[select-key ,(1st args)]]
264          ,(selectq* (rest args)))))
265
266   (define SELECTQ*
267     (lambda simple [cases]
268       (cond [(empty cases) `[]]
269             [(= (1st (1st cases)) 'ST)
270              (2nd (1st cases))]
271             [ST `(if (= select-key ,'(1st (1st cases)))
272                      (block . ,(rest (1st cases)))
273                      ,(selectq* (rest cases)))]])))
274
275   (define SELECT
276     (lambda macro args
277       `(let [[select-key ,(1st args)]]
278          ,(select* (rest args)))))
279
280   (define SELECT*
281     (lambda simple [cases]
282       (cond [(empty cases) `[]]
283             [(= (1st (1st cases)) 'ST)
284              (2nd (1st cases))]
285             [ST `(if (= select-key ,(1st (1st cases)))
286                      (block . ,(rest (1st cases)))
287                      ,(select* (rest cases)))]])))
288
289   (define BLOCK (lambda macro args (block* args)))
290
291   (define BLOCK*
292     (lambda simple [args]
293       (cond [(empty args) (error 'too-few-args-to-block)]
294             [(unit args) (1st args)]
295             [ST `((lambda simple ?
296                      ,(block* (rest args)))
297                   ,(1st args))]])))
298
299   (define COND (lambda macro args (cond* args)))
300
301   (define COND*                              ; COND* cannot itself use COND
302     (lambda simple [args]
303       (if (empty args) `[]
304           `(if ,(1st (1st args))
305                ,(2nd (1st args))
306                ,(cond* (rest args))))))
```

```
307
308   (define AND
309     (lambda macro args
310       (if (rail args) (and* args) `!(and* ↑,args))))
311
312   (define AND*
313     (lambda simple [args]
314       (if (empty args)
315           '$T
316           `(if ,(1st args) ,(and* (rest args)) $F))))
317
318   (define OR
319     (lambda macro args
320       (if (rail args) (or* args) `!(or* ↑,args))))
321
322   (define OR*
323     (lambda simple [args]
324       (if (empty args) '$F `(if ,(1st args) $T ,(or* (rest args))))))
326
326   (define MAP
327     (lambda simple args
328       (map* (1st args) (rest args))))
329
330   (define MAP*
331     (lambda simple [fun vectors]
332       (if (empty vectors)
333           (fun)
334           (if (empty (1st vectors))
335               (1st vectors)
336               (prep (fun . (firsts vectors))
337                     (map* fun (rests vectors)))))))
338
339   (define FIRSTS
340     (lambda simple [vectors]
341       (if (empty vectors)
342           vectors
343           (prep (1st (1st vectors))
344                 (firsts (rest vectors))))))
345
346   (define RESTS
347     (lambda simple [vectors]
348       (if (empty vectors)
349           vectors
350           (prep (rest (1st vectors))
351                 (rests (rest vectors))))))
352
353   (define PROTECTING
354     (lambda macro [names body]
355       `(let ,(protecting* names) ,body)))
356
357   (define PROTECTING*
358     (lambda simple [names]
359       (if (empty names)
360           `[]
361           (prep `[,(1st names) ,(1st names)]
362                 (protecting* (rest names))))))
363   ]))
```

```
364
365   (defun 3-define-utilities-3 ()
366      (in-3-lisp \[
367
368   (define REBIND
369     (lambda simple [var binding env]
370        (if (normal binding)
371            (rebind* var binding env)
372            (error 'binding-is-not-in-normal-form))))
373
374   (define REBIND*
375     (lambda simple [var binding env]
376        (cond [(empty env) (rplact 0 †env †[[var binding]])]
377              [(= var (1st (1st env)))
378               (rplacn 2 †(1st env) †binding)]
379              [$T (rebind* var binding (rest env))])))
380
381   (define SET
382     (lambda reflect [[var binding] env cont]
383        (normalise binding env
384           (lambda simple [binding*]
385              (cont (*rebind var binding* env))))))
386
387   (define DEFINE
388     (protecting [z]
389        (lambda macro [label form]
390           `(set ,label (,†z (lambda simple [,label] ,form))))))
391
392   (define ERROR
393     (lambda reflect [a e c]
394        (undefined)))
395
396   ]))
```

Symbol Table for: 3-LISP.LSP[1,1634]

```
1ST ..................... DEFMACRO 002 018
1ST ..................... DEFINE   015 082
2ND ..................... DEFMACRO 002 019
2ND ..................... DEFINE   015 083
3-APPLY ................. EXPR ... 006 019
3-ARGUMENT-CHECK ........ EXPR ... 006 128
3-ATOM .................. EXPR ... 009 019
3-ATOM-CHECK ............ EXPR ... 009 027
3-BACKQ-MACRO ........... EXPR ... 003 127
3-BIND .................. DEFMACRO 002 029
3-BIND* ................. EXPR ... 006 020
3-BINDING ............... EXPR ... 006 008
3-BOOL-CHECK ............ EXPR ... 009 032
3-BOOLEAN ............... DEFMACRO 002 027
3-CANONICALISE .......... EXPR ... 010 009
3-CIRCULAR-CLOSURE-P .... EXPR ... 003 211
3-CIRCULAR-CLOSURES ..... EXPR ... 013 060
3-COMMA-MACRO ........... EXPR ... 003 131
3-DEFINE-REFLECTIVE ..... EXPR ... 014 006
3-DEFINE-UTILITIES-0 .... EXPR ... 015 011
3-DEFINE-UTILITIES-1 .... EXPR ... 015 077
3-DEFINE-UTILITIES-2 .... EXPR ... 015 245
3-DEFINE-UTILITIES-3 .... EXPR ... 015 365
3-DROP-LEVEL ............ EXPR ... 007 010
3-EQUAL ................. EXPR ... 010 017
3-ERROR ................. EXPR ... 012 008
3-EXPAND ................ EXPR ... 003 140
3-EXPAND-PAIR ........... EXPR ... 003 146
3-EXPAND-RAIL ........... EXPR ... 003 157
3-HANDLE ................ EXPR ... 009 024
3-HANDLE-CHECK .......... EXPR ... 009 030
3-ILLEGAL-ATOM .......... EXPR ... 012 049
3-ILLEGAL-BOOLEAN ....... EXPR ... 012 045
3-ILLEGAL-CHAR .......... EXPR ... 012 041
3-IMPLEMENTATION-ERROR .. EXPR ... 012 039
3-INCREMENT-LEVEL ....... EXPR ... 007 014
3-INDEX-ERROR ........... EXPR ... 012 036
3-INIT .................. EXPR ... 013 005
3-INITIAL-ENVIRONMENT ... EXPR ... 013 042
3-INTERRUPT-HANDLER ..... EXPR ... 012 056
3-LENGTH ................ EXPR ... 008 038
3-LENGTH* ............... DEFMACRO 002 079
3-LISP .................. EXPR ... 011 026
3-LISPIFY ............... EXPR ... 011 047
3-LOGIN ................. EXPR ... 011 017
3-LOGOUT ................ EXPR ... 011 012
3-MAKE-PRIMITIVE-CLOSURE EXPR ... 013 055
3-NORMAL ................ EXPR ... 010 042
3-NORMALISE ............. EXPR ... 004 015
3-NORMALISE* ............ DEFMACRO 002 046
3-NORMALISE-RAIL ........ EXPR ... 004 032
3-NTH ................... EXPR ... 008 046
3-NTHCDR* ............... EXPR ... 008 057
3-NUM-CHECK ............. EXPR ... 009 031
3-NUMERAL ............... DEFMACRO 002 026
3-PAIR .................. EXPR ... 009 021
3-PAIR-CHECK ............ EXPR ... 009 029
3-PNORMAL ............... EXPR ... 010 045
3-PREP .................. EXPR ... 008 034
3-PRIMITIVE-REDUCE-SIMPLE EXPR ... 004 046
3-PRIMITIVE-SIMPLE-ID ... DEFMACRO 002 024
3-PRINT ................. EXPR ... 003 173
3-PRINT-ELEMENTS ........ EXPR ... 003 196
3-PROC-TYPE ............. EXPR ... 009 067
3-PROMPT ................ EXPR ... 003 206
3-QUIT-INTERRUPT ........ EXPR ... 012 086
3-RAIL .................. EXPR ... 009 020
3-REF-TYPE .............. EXPR ... 009 037
3-REF-TYPE-ERROR ........ EXPR ... 012 031
3-RPLACN ................ EXPR ... 008 053
3-RPLACT ................ EXPR ... 008 090
3-RS .................... DEFMACRO 008 023
3-SCONS ................. EXPR ... 008 016
3-STRIP ................. DEFMACRO 002 067
3-STRIP* ................ DEFMACRO 002 071
3-TAIL .................. EXPR ... 008 042
3-TAIL* ................. EXPR ... 008 066
3-TYPE .................. EXPR ... 009 007
3-TYPE-ERROR ............ EXPR ... 012 026
3A-1ST .................. DEFMACRO 005 014
3A-2ND .................. DEFMACRO 005 015
3A-3RD .................. DEFMACRO 005 016
3A-4TH .................. DEFMACRO 005 017
3A-ARG .................. DEFMACRO 005 013
3A-ENV .................. DEFMACRO 005 012
3R-1ST .................. DEFMACRO 002 052
3R-2ND .................. DEFMACRO 002 053
3R-3RD .................. DEFMACRO 002 054
3R-4TH .................. DEFMACRO 002 055
3RD ..................... DEFMACRO 002 020
3RD ..................... DEFINE   015 084
4TH ..................... DEFINE   015 085
AND ..................... DEFINE   015 308
AND* .................... DEFINE   015 312
APPEND .................. DEFINE   015 202
ATOM .................... DEFINE   015 094
BIND .................... DEFINE   015 136
BINDING ................. DEFINE   015 116
BLOCK ................... DEFINE   015 289
BLOCK* .................. DEFINE   015 291
BODY .................... DEFINE   015 124
BOOLEAN ................. DEFINE   015 099
CDADR ................... DEF .... 013 057
COND .................... DEFINE   015 299
COND* ................... DEFINE   015 301
COPY .................... DEFINE   015 192
DEFINE .................. DEFINE   015 387
DOUBLE .................. DEFINE   015 092
EMPTY ................... DEFINE   015 090
ENV ..................... DEFINE   015 122
ERROR ................... DEFINE   015 392
FIRSTS .................. DEFINE   015 339
FOOT .................... DEFINE   015 088
FUNCTION ................ DEFINE   015 105
HANDLE .................. DEFINE   015 098
ID ...................... DEFINE   015 080
IF ...................... DEFINE   015 150
IN-3-LISP ............... DEFMACRO 002 034
JOIN .................... DEFINE   015 196
LET ..................... DEFINE   015 248
LET* .................... DEFINE   015 263
LIST? ................... DEFMACRO 002 017
LOOP-CATCH .............. DEFMACRO 011 005
MACRO ................... DEFINE   015 229
MAKE-C1 ................. DEFINE   014 031
MAP ..................... DEFINE   015 326
MAP* .................... DEFINE   015 330
MATCH ................... DEFINE   015 140
MEMBER .................. DEFINE   015 166
NORMAL .................. DEFINE   015 179
NORMALISE ............... DEFINE   014 016
NORMALISE-RAIL .......... DEFINE   014 041
NOT ..................... DEFINE   015 190
NUMBER .................. DEFINE   015 101
```

Symbol Table for: 3-LISP.LSP[1,1634]

3-RAIL-CHECK	EXPR	009	028	NUMERAL	DEFINE	015	097
3-RCONS	EXPR	008	010	OR	DEFINE	015	316
3-READ	EXPR	003	084	OR*	DEFINE	015	322
3-READ*	EXPR	003	087	PAIR	DEFINE	015	096
3-READ-BOOLEAN	EXPR	003	096	PATTERN	DEFINE	015	123
3-READ-PAIR	EXPR	003	102	POP	DEFINE	015	222
3-READ-RAIL	EXPR	003	113	PREP*	DEFINE	015	172
3-REBIND	EXPR	006	046	PRIMITIVE	DEFINE	015	107
3-REDUCE	EXPR	004	021	PROCEDURE-TYPE	DEFINE	015	126
3-REF	EXPR	009	064	PROMPT	DEFINE	015	114
PROTECTING	DEFINE	015	353	SELECTQ*	DEFINE	015	266
PROTECTING*	DEFINE	015	357	SEQUENCE	DEFINE	015	102
PUSH	DEFINE	015	214	SET	DEFINE	015	381
RAIL	DEFINE	015	095	SMACRO	DEFINE	015	235
READ-NORMALISE-PRINT	DEFINE	014	008	TRUTH-VALUE	DEFINE	015	103
REBIND	DEFINE	015	368	UNIT	DEFINE	015	091
REBIND*	DEFINE	015	374	XCONS	DEFINE	015	132
REDIRECT	DEFINE	015	206	~3-BACKQUOTE	DEFMACRO	002	040
REDUCE	DEFINE	014	023	~C0~	EXPR	005	031
REST	DEFINE	015	087	~C1~	EXPR	005	060
RESTS	DEFINE	015	346	~C2~	EXPR	005	091
SELECT	DEFINE	015	275	~C3~	EXPR	005	103
SELECT*	DEFINE	015	280	~C4~	EXPR	005	114
SELECTQ	DEFINE	015	261	~C5~	EXPR	005	124

Cref of: 3-LISP.LSP[1,1634]

Key to types of symbol occurrences (Note references come last):

Dash - Reference. f - Function. b - Bound. = - Top-level Setq.
t - Prog tag. c - Catch tag. p - Property name. m - Macro.
l - Lap tag. a - Array. @ - @define. d - Defprop (or @define'd definer).

&OPTIONAL 003b084 003b087 012b008
1ST 002d018 015d082 005-042 014-035 014-045 015-082 015-119 015-119 015-119
015-122 015-134 015-147 015-147 015-169 015-175 015-177 015-196
015-220 015-226 015-250 015-257 015-257 015-259 015-263 015-269
015-269 015-270 015-271 015-271 015-272 015-277 015-283 015-283
015-284 015-285 015-285 015-286 015-294 015-297 015-304 015-304
015-305 015-316 015-324 015-328 015-334 015-335 015-343 015-343
015-350 015-361 015-361 015-377 015-377 015-378
2ND 002d019 015d083 005-042 014-035 015-083 015-119 015-123 015-251 015-269
015-270 015-284 015-305
3-APPLY 005f019 004-016 004-017 004-034 004-050 005-070 005-105
3-ARGUMENT-CHECK 005f128 005-064
3-ATOM 009f019 003-219 004-016 009-027
3-ATOM-CHECK 009f027 006-009 006-047
3-BACKQ-MACRO 003f127 003-072
3-BIND 002d029 005-050 005-085
3-BIND* 006f020 002-030 006-024 006-033
3-BINDING 006f008 004-016 005-119 013-021 013-022 013-025 013-026 013-028
3-BOOL-CHECK 009f032 004-055
3-BOOLEAN 002d027 009-032
3-CANONICALISE 010f009 010-020 010-021
3-CIRCULAR-CLOSURE-P 003f211 003-186
3-CIRCULAR-CLOSURES 013f060 013-045
3-COMMA-MACRO 003f131 003-073
3-DEFINE-REFLECTIVE 014f005 013-024
3-DEFINE-UTILITIES-0 015f011 013-023
3-DEFINE-UTILITIES-1 015f077 013-027
3-DEFINE-UTILITIES-2 015f245 013-029
3-DEFINE-UTILITIES-3 015f365 013-030
3-DROP-LEVEL 007f010 005-069 005-072 005-077 005-118
3-EQUAL 010f017 004-069 010-028 010-034
3-ERROR 012f008 002-011 003-132 005-132 006-012 006-030 006-031 006-040 006-041
006-050 008-079 009-047 009-048 010-036 012-027 012-032 012-037
012-039 012-042 012-046 012-050
3-EXPAND 003f140 002-040 003-149 003-150 003-151 003-159
3-EXPAND-PAIR 003f146 003-142
3-EXPAND-RAIL 003f157 003-143
3-HANDLE 009f024 003-152 003-152 003-161 003-218 003-220 009-030 009-055 009-060
010-026 010-027 010-043
3-HANDLE-CHECK 009f030
3-ILLEGAL-ATOM 012f049 003-092
3-ILLEGAL-BOOLEAN 012f045 003-100
3-ILLEGAL-CHAR 012f041 003-090 003-108
3-IMPLEMENTATION-ERROR 012f039 004-080
3-INCREMENT-LEVEL 007f014 005-023 005-047
3-INDEX-ERROR 012f036 008-058 008-063 008-067 008-093 008-099
3-INIT 013f005 002-035 011-018 011-030
3-INITIAL-ENVIRONMENT 013f042 013-015
3-INTERRUPT-HANDLER 012f056 011-029
3-LENGTH 008f038 003-212 003-214 003-217 004-059 005-131 005-131
3-LENGTH* 002d079 008-039 008-040
3-LISP 011f026 012-019 012-083
3-LISPIFY 011f047 002-037
3-LOGIN 011f017
3-LOGOUT 011f012
3-MAKE-PRIMITIVE-CLOSURE 013f055 013-044
3-NORMAL 010f042 004-017 006-049 010-053 010-054 010-055 010-059
3-NORMALISE 004f015 011-041 011-054
3-NORMALISE* 002d046 004-022 004-035 004-049 005-037 005-049 005-073 005-083 005-119
3-NORMALISE-RAIL 004f032 004-018 005-093
3-NTH 008f046 004-060

3-NTHCDR* 008f057 008-047 008-048 008-054
3-NUM-CHECK 009f031 004-073 004-073 004-074 004-074 004-075 004-075 004-078 004-076
3-NUMERAL 002d026 009-031
3-PAIR 009f021 009-029
3-PAIR-CHECK 009f029 004-057 004-058 004-065 004-066 009-068
3-PNORMAL 010f045 010-043
3-PREP 008f034 004-062
3-PRIMITIVE-REDUCE-SIMPLE 004f046 005-055
3-PRIMITIVE-SIMPLE-ID 002d024 005-065
3-PRINT 003f173 003-180 003-184 003-190 003-204 004-071 005-116 012-016
3-PRINT-ELEMENTS 003f196 003-188 003-189 003-193
3-PROC-TYPE 009f067 005-036
3-PROMPT 003f206 005-115 005-117 011-037
3-QUIT-INTERRUPT 012f086 012-066 012-072
3-RAIL 009f020 003-185 003-213 003-216 004-018 005-130 006-032 009-028 009-056
010-023 010-024 010-052 010-056
3-RAIL-CHECK 009f028 004-047 004-063 004-064 006-010 006-048 008-011 008-016 008-035
008-039 008-043 008-047 008-054 008-091 008-092
3-RCONS 008f010 004-063
3-READ 003f084 002-011 003-059 004-070 005-119 011-038
3-READ* 003f087 002-011 003-066 003-068 003-069 003-085 003-103 003-104 003-106
003-109 003-115 003-115
3-READ-BOOLEAN 003f096 003-093
3-READ-PAIR 003f102 003-070
3-READ-RAIL 003f113 003-071
3-REBIND 006f046 004-078
3-REDUCE 004f021 004-019 005-024 005-078
3-REF 009f054 003-061
3-REF-TYPE 009f037 004-054 012-033
3-REF-TYPE-ERROR 012f031 008-027 009-061 009-062
3-RPLACN 008f053 004-067 006-054 013-047 013-049
3-RPLACT 008f090 004-068
3-RS 008d023 008-035 008-039 008-043 008-047
3-SCONS 008f015 004-064
3-STRIP 002d067 002-052 002-053 002-053 002-054 002-054 002-054 002-055 002-055
002-055 002-055 002-081 002-081 003-158 003-158 003-197 003-198
003-198 004-033 006-011 006-011 006-025 006-025 006-026 006-026
006-035 006-035 006-036 006-036 008-011 008-011 008-016 008-016
008-060 008-060 009-057 009-057 010-031 010-031 010-032 010-032
010-057 010-057
3-STRIP* 002d071 006-051 006-051 008-071 008-071 008-096 008-096 010-025 010-025
3-TAIL 008f042 004-061
3-TAIL* 008f066 005-094 008-043 008-044
3-TYPE 009f007 003-141 003-174 006-021 006-023 008-024 009-021 009-038 009-042
010-018 010-018 010-019 012-028
3-TYPE-ERROR 012f026 006-042 006-043 006-044 009-027 009-028 009-029 009-030 009-031
009-032 009-072
3A-1ST 005d014 005-062 005-094 005-106
3A-2ND 005d015 005-033 005-095
3A-3RD 005d016 005-034 005-069 005-072 005-077
3A-4TH 005d017 005-035 005-067 005-069 005-072 005-077 005-086 005-106
3A-ARG 005d013
3A-ENV 005d012 005-020
3R-1ST 002d052 003-213 003-214 003-215 003-215 003-218 003-219 004-035 004-049
004-054 004-055 004-056 004-057 004-058 004-059 004-060 004-061
004-062 004-065 004-066 004-067 004-068 004-069 004-071 004-073
004-074 004-075 004-076 004-077 004-078 005-012 005-014 005-052
005-070 005-073 005-078 005-085 006-013 006-053 010-053 013-049
3R-2ND 002d053 003-220 003-221 004-049 004-055 004-056 004-060 004-061 004-062
004-065 004-066 004-067 004-068 004-069 004-073 004-074 004-075
004-076 004-078 005-013 005-014 005-015 005-016 005-016 005-017
005-050 005-074 005-079 005-085 005-129 006-013 010-054 013-047
013-058
3R-3RD 002d054 002-024 004-055 004-067 004-068 004-078 005-016 005-049 005-075
005-080 005-084 005-133 010-055 013-058
3R-4TH 002d055 005-017 005-081
3RD 002d020 015d084 015-084 015-124
4TH 015d085

A 003b114 003-116
AND 015d308 003-152 003-161 003-161 003-212 003-216 005-130 006-028 006-038 009-019 009-020 009-024 010-010 010-018 010-023 010-026 010-048 010-056 012-058 015-144 015-186 015-187
AND* 015d312 015-310 015-310 015-316
APPEND 015d202
ARGS 004b021 004b046 005b128 008b010 008b015 008b011 008b016 004-023 004-026 004-027
ARGS* 005b060 005-064 005-066 005-070 005-073 005-074 005-075 005-078 005-079 005-080 005-081 005-085
ATOM 012b049 015d094 012-050 012-050
B 003b109 003b115 003-110 003-111
BIND 015d136 005-045 014-038
BINDING 006b046 015d116 006-049 006-050 006-050 006-054 006-055
BINDS 006b024 006b033 006-029 006-033 006-034 006-039
BLOCK 015d289 007-022 014-010 015-114 015-272 015-286
BLOCK* 015d291 015-289 015-296
BODY 015d124 002-034 002-036 005-044 011-005 013-085 013-086 014-037 015-233 015-241 015-250 015-256 015-258 015-355
BOOLEAN 015d099 003-176 009-009 009-032 009-040 010-029
C 003b110 003-111
CDADR 013d057
CHAR 012b041 012-042
CHARACTER 012b056 012-057 012-061
COND 015d299 003-089 003-098 003-147 003-181 004-016 005-041 005-063 006-028 006-038 009-008 009-055 009-069 014-018 014-034 015-118 015-142 015-168 015-174 015-268 015-282 015-293 015-376
COND* 015d301 015-299 015-306
CONT 004b015 004b021 004b032 004b046 005b019 005b031 005b060 005b091 005b103 005b114 005b124 007b010 004-016 004-017 004-018 004-019
COPY 015d192 015-196 015-204
DEFINE 015d387 014-008 014-016 014-023 014-031 014-041 015-065 015-080 015-082 015-083 015-084 015-085 015-087 015-088 015-090 015-091 015-092 015-094 015-095 015-095 015-097 015-098 015-099 015-101 015-102 015-103 015-105 015-107 015-114 015-116 015-122 015-123 015-124 015-126 015-132 015-136 015-140 015-150 015-166 015-172 015-179 015-190 015-192 015-198 015-202 015-206 015-214 015-222 015-229 015-236 015-248 015-253 015-261 015-266 015-275 015-280 015-289 015-291 015-299 015-301 015-308 015-312 015-318 015-322 015-326 015-330 015-339 015-346 015-353 015-367 015-368 015-374 015-381 015-392
DOUBLE 015d092 015-176
E 009b019 009b020 009b021 009b024 009b027 009b028 009b029 009b030 009b031 009b032
E1 010b017 010b031 010-018 010-019 010-029 010-030 010-031 010-031 010-031 010-033 010-034
E2 010b017 010b032 010-018 010-029 010-030 010-032 010-032 010-032 010-033 010-034
EL 008b034 008b053 008-036
ELEMENT* 005b091 005-096 005-098
ELEMENTS 003b159 009b058 003-161 003-162 003-163
EMPTY 015d090 014-043 015-118 015-144 015-144 015-145 015-146 015-168 015-174 015-194 015-218 015-224 015-255 015-268 015-282 015-293 015-303 015-314 015-324 015-332 015-334 015-341 015-348 015-359 015-376
ENTRY 013b055 013-056 013-057
ENV 004b015 004b021 004b032 005b031 005b060 005b091 005b103 005b114 005b124 006b008 006b046 007b010 006b011 006b051 015d122 004-016 004-018 004-019
ERROR 015d392 015-045 015-118 015-145 015-146 015-174 015-209 015-226 015-293 015-372
EXP 003b173 003b211 004b015 008b034 008b038 008b042 008b046 009b007 009b037 009b054 010b009 010b042 010b045 012b026 012b031 012b045 010b057 003-174 003-175 003-176 003-177 003-179 003-181 003-182 003-184 003-185 003-186 003-188 003-189 003-190 003-193
EXPR 011b047 012b008 011-050
EXPRS 002b036 002-037 002-038
F 003b140 003b146 003b157 003-142 003-143
FIRSTS 015d339 015-336 015-344
FLAG 003b196 003b199 003-199 003-199 003-203

FOOT 015d088
FORMS 002b037 002-038
FUNCTION 015d105 009-046
HANDLE 015d098 003-180 006-032 008-025 009-012 009-030 009-042 010-020 015-143
HEADS 008b101 008-102
I 008b059 008b070 008b094 008-061
ID 015d080 013-028 014-011
IF 015d150 003-105 003-108 003-132 003-152 003-161 003-177 003-185 003-186
003-201 003-203 004-033 004-048 004-055 004-069 005-021 005-130
006-013 006-032 006-049 006-053 007-016 008-058 008-062 008-067
008-068 008-073 008-074 008-093 008-099 008-100 008-105 009-027
009-028 009-029 009-030 009-031 009-032 009-043 009-060 010-010
010-034 010-059 012-011 012-015 012-017 012-079 012-087 014-043
015-190 015-194 015-208 015-218 015-224 015-255 015-271 015-285
015-303 015-304 015-310 015-314 015-316 015-320 015-324 015-324
015-332 015-334 015-341 015-348 015-359 015-370
IGNORE 012b056
IN-3-LISP 002d034 014-006 015-012 015-078 015-246 015-366
JOIN 015d198 015-138 015-147 015-204
LAST 008b095 008-103
LET 015d248 003-058 003-064 003-065 003-085 003-088 003-097 003-103 003-128
003-133 003-150 003-197 003-215 005-020 005-023 005-033 005-047
005-062 005-082 005-129 008-076 008-100 010-020 011-033 013-043
013-056 014-011 015-263 015-277 015-365
LET* 015d253 015-258
LEVEL 003b206 003-208
LIST 003b196 003b198 003-198 003-198 003-198 003-200 003-201 003-204
LIST? 002d017 009-020 009-024 010-048
LOOP-CATCH 011d005 011-032 011-036 011-041 011-054
MACRO 015d229 015-249 015-254 015-262 015-276 015-289 015-299 015-309 015-319
015-354 015-389
MAKE-C1 014d031 004-027 014-029
MAP 015d326 015-143 015-186 015-250 015-251
MAP* 015d330 015-328 015-337
MATCH 015d140 015-138 015-143 015-147 015-148
MEMBER 015d166 015-109 015-170 015-187
MESSAGE 012b008 012-011 012-012 012-014
N 008b042 008b046 008b053 008b057 008b066 008b090 012b036 002b080 008-043
008-044
NEW 008b012 008b017 008-013
NORMAL 015d179 006-050 009-047 009-048 014-018 015-186 015-188 015-370
NORMAL-FORM 005b019 005b114 005b124 005-022
NORMALISE 014d016 004-027 005-042 005-044 013-025 014-011 014-025 014-029 014-035
014-037 014-045 015-155 015-158 015-159 015-162 015-233 015-239
015-241 015-383
NORMALISE-RAIL 014d041 004-038 014-020 014-047
NOT 015d190 002-069 002-073 003-108 005-131 006-049 006-050 007-016 008-099
008-102 008-104 009-019 009-047 009-048 009-060 010-034 010-059
NUMBER 015d101 009-039
NUMERAL 015d097 003-175 009-008 009-031 009-039 010-030
O-RAIL 008b066 008-067 008-069 008-071 008-079
OK 003b087 003-089
OR 015d318 002-035 003-091 006-027 006-037 008-061 008-072 008-097 009-043
010-022 010-043 010-046 010-049 011-018 011-030
OR* 015d322 015-320 015-320 015-324
PAIR 015d096 003-142 003-181 009-013 009-021 009-029 009-043 013-064 013-065
013-076 013-077 014-021 015-187 015-188
PATTERN 006b020 006b025 006b035 015d123 006-021 006-024 006-025 006-025 006-025
006-027 006-028 006-033 006-035 006-035 006-035 006-037 006-038
006-044
POP 015d222 007-017
PREP* 015d172 015-177
PRIMITIVE 015d107 005-043 014-036
PROC 004b021 004b046 005b031 006b128 009b067 004-022 004-023
PROCEDURE-TYPE 015d126 004-025 014-027
PROMPT 015d114 007-022 014-010 014-012
PROTECTING 015d353 015-338
PROTECTING* 015d357 015-355 015-362
PUSH 015d214 007-011

R1 008b090 008-091 008-093 008-095 008-096 008-099 008-107
R1-HEADER 008b103 008-104 008-105
R2 008b090 008b100 008-092 008-100 008-100 008-100 008-101 008-102 008-106
RAIL 003b157 004b032 008b053 008b057 012b036 002b081 003b158 008b071 008b096
009b057 015d095 003-158 003-158 003-158 003-159 003-160
READ-NORMALISE-PRINT 014d008 007-024 014-014
REBIND 015d368
REBIND* 015d374 015-371 015-379
REDIRECT 015d206
REDUCE 014d023 013-026 014-021
REST 002b068 002b072 008b060 015d087 002-069 002-069
REST* 005b103 005-105
RESTS 015d346 015-337 015-351
SELECT 015d275 015-128
SELECT* 015d280 015-278 015-287
SELECTQ 015d261 004-025 012-061 014-027 015-181
SELECTQ* 015d266 015-264 015-273
SEQUENCE 015d102 009-041
SET 015d381 015-390
SMACRO 015d235
STREAM 003b084 003b087 003b096 003b102 003b113 003b127 003b131 003-085
TAG 012b086 012-090
TRUTH-VALUE 015d103 009-040
TYPE 012b026 012b031 012-027
UNIT 015d091 015-175 015-294
VALS 006b020 006b026 006b036 006-023 006-024 006-026 006-026 006-026 006-027
006-028 006-030 006-033 006-036 006-036 006-037 006-038 006-040
006-042 006-043
VAR 006b008 006b046 006-009 006-012 006-013
X 003b140 003b146 003-141 003-142 003-143
XCONS 015d132
~3-BACKQUOTE 002d040 003-129 003-148
~C0~ 005f031 013-011
~C1~ 005f060 013-011
~C2~ 005f091 013-011
~C3~ 005f103 013-011
~C4~ 005f114 013-011
~C6~ 005f124 013-011

Notes

Preface and Prologue

1. Bobrow and Winograd (1977), and Bobrow et al. (1977)
2. Weyhrauch (1978), Doyle (1979), McCarthy (1968), Hayes (1979), and Davis (1980a), respectively.
3. For a discussion of macros see the various sources on LISP mentioned in note 16 of chapter 1; meta-level rules in representation were discussed in Brachman and Smith (1980); for a collection of papers on non-monotonic reasoning Bobrow (1980); macros are discussed in Pitman (1980).
4. Brachman (1980).
5. Newell and Simon (1963); Newell and Simon (1956).
6. The proceduralist view was represented particularly by a spate of dissertations emerging from MIT at the beginning of the 1970s; see for example Winograd (1972), Hewitt (1972), Sussman et al. (1971), etc.
7. See Minsky (1975), Winograd (1975), and all of the systems reported in Brachman and Smith (1980).
8. Searle (1980), Fodor (1978 and 1980).
9. Brachman and Smith (1980).
10. See the introduction to Brachman and Smith (1980).
11. References on *node, frame, unit, concept, schema, script, pattern, class,* and *plans* can be found in the various references provided in Brachman and Smith (1980).
12. See in particular Hayes (1978).
13. The distinction between central and peripheral aspects of mind is articulated in Nilsson (1981); on the impossibility of central AI (Nilsson himself feels that the central faculty will quite definitely succumb to AI's techniques) see Dreyfus (1972) and Fodor (1980 and forthcoming).
14. Nilsson (1981).

Chapter 1 (Introduction)

1. PROLOG has been presented in a variety of papers; see for example Clark and McCabe (1979), Roussel (1975), and Warren et al. (1977). The conception of logic as a programming language (with which we radically disagree) is presented in Kowalski (1974 and 1979).

2. For a discussion of the semantical properties of computational systems see for example Fodor (1980), Fodor (1978), and Haugeland (1978).
3. Such facilities as provided in MDL are described in Galley and Pfister (1975); in INTERLISP, in Teitelman (1978).
4. Reiter (1978), McDermott and Doyle (1978), Bobrow (1980).
5. Clark and McCabe (1979).
6. McCarthy et al. (1965).
7. Sussman and Steele (1975); Steele and Sussman (1978a).
8. Greiner and Lenat (1980), Genesereth and Lenat (1980).
9. Quine (1953a), p. 79 in the 1963 edition.
10. References on the message-passing metaphor. See Hewitt et al. (1974), Hewitt (1977), for ACT1 see Lieberman (19??); SMALLTALK see Goldberg (1981), Ingalls (1978).
11. Fodor (forthcoming)
12. See, however, the postscript, where we in part disavow this fractured notion of syntactic and semantic domains.
13. Fodor (1980).
14. Gordon (1973 and 1975);
15. Church (1941).
16. SCHEME is reported in Sussman and Steele (1975) and in Steele and Sussman (1978a); MDL in Galley and Pfister (1975), NIL in White (1979), MACLISP in Moon (1974) and Weinreb and Moon (1981), and INTERLISP in Teitelman (1978). COMMON LISP and SEUS are both under development and have not yet been reported in print, so far as we know (personal communication with Guy and with Richard Weyhrauch).
17. Stallman and Sussman (1977), deKleer et al. (1977).
18. Davis (1980)
19. Stefik (19..1b).
20. deKleer et al. (1977).
21. Doyle (1981).
22. References to specific LISP dialects are given in note 15, above; more general accounts may be found in Allen (1978), Weisman (1967), Winston and Horn (1981), Charniak et al. (1980), McCarthy et al. (1965), and McCarthy and Talbott (forthcoming).
24. Clark and McCabe (1979), Roussel (1975), and Warren et al. (1977).
25. Goldberg (1981); Ingalls (1978).
26. Weyhrauch (1978).
27. I am indebted here to Richard Weyhrauch for personal communication on these points.

Chapter 2 (1-LISP: A Basis Dialect)

1. It is reported that Jean Sammett introduced a conference on programming languages with the comment that modern programming languages could be divided into two large classes: LISP, and the rest.
2. Gordon (1973, 1975a, 1975b, and 1979).
3. Moses (1970) and Steele and Sussman (1978b).
4. Quite understandably, there are differences between the SCHEMES reported in Sussman and Steele (1975) and Steele and Sussman (1978b), and between either of these and the current implementation to be found on the PDP-10 at the M.I.T. Artificial Intelligence Laboratory.
5. Moses (1970).
6. Pitman (1980).
7. References are given in notes 22 and 23 of chapter 1, above.
8. Steele and Sussman (1978b).
9. Maturana (1978).

Chapter 3 (Semantic Rationalisation)

1. Clark and McCabe (1979), Roussel (1975), and Warren et al. (1977).
2. Gordon (1975a, 1975b).
3. Tennent (1976), Gordon (1979), Stoy (1977), etc.
4. Gordon (1979), p. 35.
5. Tarski (1936 and 1944).
6. Weinreb and Moon (1981).
7. Donnellan (1966).
8. Sussman and Steele (1980), and Steele (1980).
9. Searle (1969).
10. Winograd (1975).
11. Quine (1951).
12. Frege (1884), p. X.
13. Tarksi (1936)
14. McCarthy et al. (1965).
15. Quine (1953b).

Chapter 4 (2-LISP: A Rationalised Dialect)

1. Weinreb and Moon (1981).
3. Steele and Sussman (1978b).
4. Montague (1970, 1973).
5. Lewis (1972).
6. Rogers (1967), Kleene (1952).
7. Quine (1966).
8. Quine (1978).
9. Montague (1973); see for example p. 257 in the version printed in Thomason (1974).
10. Steele and Sussman (1978b).

Chapter 5 (Procedural Reflection and 3-LISP)

1. Steele and Sussman (1978b) pp. 47-50.

Chapter 6 (Conclusion)

1. In preliminary conversations about these issues Gerry Sussman has suggested that this proposal — that the evaluator *always* de-reference expressions — best reconstructs his understanding of how LISP should be designed and/or described. There is some evidence (see for example Steele and Sussman (1978b) p. 10) that his comment is true to the conception of LISP embodied in SCHEME; see, however, the subsequent discussion.

References

Allen, J., *Anatomy of LISP*, New York: McGraw-Hill (1978).

Bobrow D. G., (ed.) *Artificial Intelligence*, 13:1,2 (Special Issue on Non-Monotonic Reasoning), (1980).

Bobrow, D. G., and Winograd, T., "An Overview of KRL: A Knowledge Representation Language", *Cognitive Science* 1:3-46 (1977)

Bobrow, D. G., Winograd, T., et al., "Experience with KRL-0: One Cycle of a Knowledge Representation Language", *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, Cambridge, Mass (August 1977) pp. 213-222.

Bobrow, D. G., and Wegbreit, B., "A Model and Stack Implementation of Multiple Environments", *Communications of the ACM 16*, 10:591-603 (Oct. 1973).

Brachman, R. "Recent Advances in Representation Languages", invited presentation at the First Annual National Conference on Artificial Intelligence, Stanford, California, (August 1980), sponsored by the American Association for Artificial Intelligence.

Brachman, R., and Smith, B. C., (eds.), *Special Issue on Knowledge Representation, SIGART Newsletter*, 70 (February 1980).

Charniak, E., Reisbeck, C., and McDermott, D., *Artificial Intelligence Programming*, Hillsdale, N.J.: Lawrence Erlbaum (1980).

Church, A., *The Calculi of Lambda-conversion*, Annals of Mathematics Studies 6, Princeton, N.J.: Princeton University Press (1941).

Clark, K. L., and McCabe, F., "Programmers' Guide to IC-Prolog", CCD Report 79/7, Imperial College, London (1979).

Davis, R. "Applications of Meta Level Knowledge to the Construction, Maintenance, and Use of Large KNowledge Bases", Ph.D. thesis, Stanford University, Stanford, California; also in Davis, R., and Lenat, D., (eds.), *Knowledge-Based Systems in Artificial Intelligence*, New York: McGraw-Hill (1980a).

———, "Meta-Rules: Reasoning about Control", M.I.T. Artificial Intelligence Laboratory Memo AIM-576 (1980b); also *Artificial Intelligence* 15:3, December 1980, pp. 179-222.

deKleer, J., Doyle, J., Steele, G. L. Jr., and Sussman, G. J., "Explicit Control of Reasoning", *Proc. of the ACM Symposium on Artificial Intelligence and Programming Languages*, Rochester, N.Y. (1977); also M.I.T. Artificial Intelligence Laboratory Memo AIM-427 (1977).

Donnellan, K., "Reference and Definite Descriptions", *Philosophical Review* 75:3 (1966), pp. 281-304.; reprinted in Rosenberg and Travis (eds.), *Readings in the Philosophy of Language*, Prentice-Hall (1971).

Doyle, J., "A Truth-Maintenance System", *Artificial Intelligence* 12:231-272 (1979).

———, *A Model for Deliberation, Action, and Introspection*, doctoral dissertation submitted to the Massachusetts Institute of Technology; also M.I.T. Artificial Intelligence Laboratory Memo AIM-TR-581 (1980).

Dreyfus, H., *What Computers Can't Do*, New York: Harper and Row (1972).

Fodor, J., *The Language of Thought*, New York: Thomas Y. Crowell, Company (1975); paperback version, Cambridge: Harvard University Press, 1979.

———, "Tom Swift and his Procedural Grandmother", *Cognition* 6 (1978); reprinted in Fodor, Jerry, *Representations*, Cambridge: Bradford, 1981.

———, "Methodological Solipsism Considered as a Research Strategy in Cognitive Psychology", *The Behavioral and Brain Sciences*, 3:1 (1980) pp. 63-73; reprinted in Haugeland (ed.), *Mind Design*, Cambridge: Bradford, 1981, and in Fodor, J., *Representations*, Cambridge: Bradford 1981.

———, *The Modularity of Mind*, Cambridge: Bradford (forthcoming).

Frege, G., *Die Grundlagen der Arithmetik, eine logisch-mathematische Untersuchung über den Begriff der Zahl* (Breslau, 1884); reprinted in *The Foundations of Arithmetic, A logico-mathematical Inquiry into the Concept of Number*, English translation by John L. Austin, Evanston, Ill.: Northwestern Univeristy Press (1950).

Galley, S. W., and Pfister, G., *The MDL Language*, Programming Technology Division Document SYS.11.01. Laboratory of Computer Science, M.I.T. (1975).

Genesereth, M, and Lenat, D. B. "Self-Description and -Modification in a Knowledge Representation Langauge", Report of the Heuristic Programming Project of the Stanford University Computer Science Dept., HPP-80-10 (1980).

Goldberg, A., et al. "Introducing the Smalltalk-80 System", and other SMALLTALK papers, *Byte* 6:8, (August 1981).

Gordon, M. J. C., "Models of Pure LISP", Dept. of Machine Intelligence, Experimental Programming Reports No., 30, University of Edinburgh (1973).

———, Operational Reasoning and Denotational Semantics", Stanford University Computer Science Dept. Deport No. STAN-CS-75-506. (1975a)

———, "Towards a Semantic Theory of Dynamic Binding", Stanford University Artificial Intelligence Laboratory, Memo 265, Stanford University Computer Science Dept. Deport No. STAN-CS-75-507 (1975b).

———, *The Denotational Description of Programming Languages: An Introduction*, New York: Springer-Verlag (1979).

Greiner, R., and Lenat, D. B., "A Representation Language Language", *Proc. of the First Annual National Conference on Artificial Intelligence*, Stanford Univ., (August 1980), pp. 165-169.

Haugeland, J., "The Nature and Plausibility of Cognitivism" *The Brain and Behavioral Sciences* 1 (1978).

Hayes, P. J., "In Defense of Logic", in *Proc. Fifth International Joint Conference on Artificial Intelligence*, Massachusetts Institute of Technology (August 1977) pp. 559-565; available from Carnegie-Mellon University, Pittsburgh, PA.

———, "The Naive Physics Manifesto", unpublished manuscript (May 1978).

———, Personal conversations on the GOLUM deduction system (1979).

Hewitt, C., "Description and Theoretical Analysis (using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot", M.I.T. Artificial Intelligence Laboratory TR-258 (1972).

———, "Viewing Control Structures as Patterns of Passing Messages", *Artificial Intelligence*, 8:3, (June 1977) pp. 324-364.

Hewitt, C., et al. "Behavioral Semantics of Nonrecursive Control Structures", *Proc. Collque sur la Programmation*, B. Robinet (ed.), in *Lecture Notes in Computer Science, No. 19*, pp. 385-407 Berlin: Springer-Verlag (1974).

Ingalls, D. H. "The Smalltalk-76 Programming System: Design and Implementation", *Conference Record of the Fifth Annual Symposium on Principles of Programming Languages*, Tucson, Arizona (January 1978) pp. 9-16.

Kleene, S. *Introduction to Metamathematics*, Princeton: D. Van Nostrand (1952).

Kowalski, R. A., "Predicate Logic as a Programming Language", *Proceedings IFIP*, Amsterdam: North Holland (1974) pp. 569-574.

Kowalski, R. A., "Algorithm = Logic + Control", *CACM* (August 1979).

Kripke, S. "Outline of a Theory of Truth", *Journal of Philosophy*, 72:690-716 (19??).

Lewis, D., "General Semantics", in Davidson and Harman (eds.), *Semantics of Natural Langauges*, Dordrecht, Holland: D. Reidel (1972), pp. 169-218.

Maturana, H., and Varela, F., *Autopoietic Systems*, in Boston studies in the philosophy of science, Boston: D. Reidel, (1978); originally issued as B.C.L. Report 9.4, Biological Computer Laboratory, University of Illinois, 1975.

McAllester, David A. "A Three-Valued Truth Maintenance System", M.I.T. Artificial Intelligence Laboratory Memo AIM-473 (1978).

McCarthy, J., "Programs with Common Sense", in M. Minsky (ed.), *Semantic Information Processing*, Cambridge: M.I.T. Press (1968), pp. 403-418.

McCarthy, J., et al., *LISP 1.5 Programmer's Manual*, Cambridge, Mass.: The MIT Press (1965).

McCarthy, J, and Talbott, C., *LISP: Programming and Proving*, Cambridge, Mass.: Bradford (forthcoming).

McDermott, D., and Doyle, J., "Non-monotonic Logic I", M.I.T. Artificial Intelligence Laboratory Memo AIM-486 (1978).

McDermott, D., and Sussman, G. "The CONNIVER Reference Manual", M.I.T. Artificial Intelligence Laboratory Memo AIM-259a, Cambridge, Mass. (1973).

Minsky, M. "Matter, Mind, and Models", in *Semantic Information Processing*, M. Minsky (ed.), Cambridge: MIT Press (1968).

Minsky, M., "A Framework for the Representation of Knowledge", in P. Winston (ed.), *The Psychology of Computer Vision*, New York: McGraw-Hill (1975) pp. 211-277.

Montague, R., "The Proper Treatment of Quantification in Ordinary English", in J. Hintikka, J. Moravevcsik, and P. Suppes (eds.), *Approcahes to Natural Language: Proceedings of the 1970 Stanford Workshop on Grammar and Semantics*, Dordrecht: Reidel (1973) pp. 221-242; reprinted in Thomason (1974).

———, "Pragmatics and Intensional Logic", *Synthese* 22 (1970) pp. 68-94; reprinted in R. H. Thomason (ed.), *Formal Philosophy: Selected Papers of Richard Montague*, New Haven: Yale Univ. Press, 1974.

Moon, D., "MacLISP Reference Manual", M.I.T. Laboratory for Computer Science, Cambridge, Mass. (1974).

Moses, J., "The Function of FUNCTION in LISP", ACM SIGSAM Bulletin, pp. 13-27, (July 1970); also M.I.T. Artificial Intelligence Laboratory Memo AIM-199 (1970).

Newell, A., and Simon, H., "The Logic Theory Machine: a complex information processing system", *IRE Transactions on Information Theory*, Vol. IT-2, No. 3, pp. 61-79.

Newell, A., and Simon, H., "GPS, a Program that Simulates Human Thought", in E. A. Feigenbaum and J. Feldman (eds.), *Computers and Thought*, New York: McGraw-Hill (1963).

Nilsson, N. "Artificial Intelligence: Engineering, Science, or Slogan?" manuscript (to be published), (April 1981).

Pitman, K., "Special Forms in LISP", *Conference Record of the 1980 LISP Conference*, Stanford University (August 1980), pp. 179-187.

Quine, W. V. O., *Mathematical Logic* [New York: Norton, 1940], Cambridge: Harvard University Press, 1947; revised edition, Cambridge, Harvard University Press (1951).

———, "Identity, Ostension, and Hypostasis", in *From a Logical Point of View*, Cambridge: Harvard University Press, (1953a); reprinted in paperback by Harper Torchbooks, 1963.

———, "On What There Is", in Quine, W. V. O., *From a Logical Point of View*, Cambridge: Harvard University Press, (1953b); reprinted in paperback by Harper Torchbooks, 1963.

———, "Three Grades of Modal Involvement", in *The Ways of Paradox, and Other Essays*, Cambridge: Harvard Univ. Press (1966).

Quine, W. V. O., and Ullian, J. S., *The Web of Belief*, New York: Randon House (1978).

Reiter, R., "On Reasoning by Default", *Proc. Second Conference on Theoretical Issues in Natural Language Processing*, University of Illinois at Champaign-Urbana (1978) pp. 210-218.

Rogers, H. Jr., *Theory of Recursive Functions and Effective Computability*, New York: McGraw-Hill (1967).

Roussel, P. "PROLOG: Manuel de Reference et d'Utilisation", Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille, Luminy (1975).

Russell, B. "Mathematical Logic as Based on the Theory of Types", *American Journal of Mathematics* 30:222-262 (1908); reprinted in Van Heijenoort, J. (ed), *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931*, Cambridge, Mass.: Harvard (1967).

Searle, J. R., *Speech Acts: An Essay in the Philosophy of Langauge*, Cambridge: Cambridge Univ. Press (1969).

———, "Minds, Brains, and Programs", *The Behavioral and Brain Sciences* 3:3 (1980) pp. 417-457; reprinted in Haugeland (ed.), *Mind Design*, Cambridge: Bradford, 1981, pp. 282-306.

Stallman, R. M., and Sussman, G. J., "Forward Reasoning and Dependency Directed Backtracking in a System for Computer-Aided Circuit Analysis", *Artificial Intelligence* 9:2 (1977) pp. 135-196; also in *Artificial Intelligence: An MIT Perspective*, Volume 1, P. H. Winston and R. H. Brown (eds.), pp. 31-91, Cambridge: M.I.T. Press (1979).

Steele, G. "LAMBDA: The Ultimate Declarative", M.I.T. Artificial Intelligence Laboratory Memo AIM-379 (1976).

———, "The Definition and Implementation of a Computer Programming Language Based on Constraints", Ph.D. Dissertation, M.I.T. Artificial Intelligence Laboratory, Report AI-TR-595 (1980).

Steele, G, and Sussman, G. "LAMBDA: The Ultimate Imperative", M.I.T. Artificial Intelligence Laboratory Memo AIM-353 (1976).

———, "The Revised Report on SCHEME, A Dialect of LISP", M.I.T. Artificial Intelligence Laboratory Memo AIM-452 (1978a).

———, "The Art of the Interpreter, or, The Modularity Complex (Parts Zero, One, and Two)", M.I.T. Artificial Intelligence Laboratory Memo AIM-453, Cambridge, Mass. (1978b).

———, "Constraints", M.I.T. Artificial Intelligence Laboratory Memo AIM-502 (1979).

Stefik, M. J., "Planning with Constraints (MOLGEN, Part 1)", *Artificial Intelligence* 16:2 (July 1981a) pp. 111-139.

———, "Planning and Meta-Planning (MOLGEN: Part 2)", *Artificial Intelligence* 16:2 (July 1981b) pp. 141-169.

Stoy, J. E., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, Cambridge: MIT Press (1977).

Sussman, G., and Steele, G., "SCHEME: An Interpreter for Extended Lambda Calculus", M.I.T. Artificial Intelligence Laboratory Memo AIM-349 (1975).

———, "CONSTRAINTS — A Language for Expressing Almost-Hierarchical Descriptions", *Artificial Intelligence* 14:1 (August 1980) pp. 1-39.

Sussman, G., et al. "Micro-PLANNER Reference Manual" M.I.T. Artificial Intelligence Laboratory Memo AIM-203a (1971).

Tarski, A., "The Concept of Truth in Formalized Languages" (1936), in Tarski, A., *Logic, Semantics, Metamathematics*, Oxford (1956).

———, "The Semantic Conception of Truth and the Foundations of Semantics", *Philosophical and Phenomenological Research* 4:341-376 (1944); reprinted in Linksy (ed.), *Semantics and the Philosophy of Language*, Urbana: University of Illinois, 1952, pp. 13-47.

Teitelman, W. "INTERLISP Reference Manual", Palo Alto: Xerox Palo Alto Research Center (1978).

Tennent, R., "The Denotational Semantics of Programming Languages", *Communications of the ACM* 19:8 pp. 437-453 (Aug. 1976).

Thomason, R., (ed.), *Formal Philosophy: Selected Papers of Richard Montague*, New Haven: Yale University Press (1974.)

Warren, D. H. D., Pereira, L. M., and Pereira, F., "PROLOG: The Language and its Implementation Compared with LISP", *Proc. Symposium on AI and Programming Languages*, Rochester, New York, *ACM SIGPLAN/SIGART Notices*, 12:8 (August 1977) pp. 109-115.

Weisman, C. *LISP 1.5 Primer*, Belmont: Dickenson Press (1967).

Weinreb, D., and Moon, D. *LISP Machine Manual*, Cambridge: Massachusetts Institute of Technology (1981).

Weyhrauch, R. W., "Prolegomena to a Theory of Mechanized Formal Reasoning", Stanford University Artificial Intelligence Laboratory, Memo AIM-315 (1978); also *Artificial Intelligence* 13:1,2 (1980) pp. 133-170.

White, J. L., "NIL — A Perspective", *Proceedings of the MACSYMA Users' Conference*, Washington, D. C. pp. 190-199 (June 1979). Available from the Laboratory for Computer Science, M.I.T., Cambridge, Mass.

Winograd, T. *Understanding Natural Language*, Academic Press (1972).

———, "Frame Representation and the Declarative-Procedural Controversy", in D. G. Bobrow and A. Collins, (eds.), *Representation and Understanding: Studies in Cognitive Science*, New York: Academic Press (1975) pp. 185-210.

Winston, P. H., and Horn, B, K. P., *LISP*, Reading, Mass: Addison-Wesley (1981).